USENIX

Proceedings of the First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA, March, 2004

conference

proceedings

First Symposium on Networked Systems Design and Implementation

San Francisco, CA, USA

March 29–31, 2004

Sponsored by

The USENIX Association

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGCOMM and ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $30 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-931971-19-6

USENIX Association

Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)

March 29–31, 2004
San Francisco, California, USA

Conference Organizers

Program Chairs

Robert Morris, *MIT*
Stefan Savage, *University of California, San Diego*

Program Committee

Brian Bershad, *University of Washington*
Bill Bolosky, *Microsoft Research*
Eric Brewer, *University of California, Berkeley*
Miguel Castro, *Microsoft Research*
Jeff Chase, *Duke University*
David Culler, *University of California, Berkeley*
Peter Druschel, *Rice University*
Dawson Engler, *Stanford University*
Steve Gribble, *University of Washington*
Butler Lampson, *MIT and Microsoft Research*
Barbara Liskov, *MIT*
Vern Paxson, *ICIR and LBL*
Jennifer Rexford, *AT&T Research*
Timothy Roscoe, *Intel Research*
Mendel Rosenblum, *Stanford University*
Ion Stoica, *University of California, Berkeley*
Marvin Theimer, *Microsoft Research*
Amin Vahdat, *Duke University*
Geoff Voelker, *University of California, San Diego*
Bill Weihl, *Akamai*

Steering Committee

Thomas Anderson, *University of Washington*
Peter Honeyman, *CITI, University of Michigan*
Mike Jones, *Microsoft Research*
Robert Morris, *MIT*
Mike Schroeder, *Microsoft*
Amin Vahdat, *Duke University*

The USENIX Association Staff

External Reviewers

Ranjita Bhagwan
Anne-Marie Kermarrec
Sean Rhea
Matt Welsh
Janet Wiener

NSDI '04: First Symposium on Networked Systems Design and Implementation

March 29–31, 2004
San Francisco, CA, USA

Index of Authors

Message from the Program Chairs

We are very pleased to introduce the proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04).

As with any new conference, the origin of NSDI can be traced to ongoing changes in the activity and direction of the research communities it serves. Historically, there have been two independent and largely disjoint communities: **networking**, focused primarily on supporting data transport between endpoints, and **distributed systems**, focused on services constructed upon an underlying communications substrate. However, in recent years members of both communities have come to appreciate the deep interaction between these concerns and the need to highlight work at the intersection of their disciplines. NSDI was created to address this need.

As NSDI's inaugural chairs, we were asked to deliver a new high-quality venue for research and experience focused on the design and implementation of networked systems. While we believe the NSDI proceedings lives up to this goal, the credit for delivering this outcome must be distributed widely. Any new conference is risky for all concerned: for the authors who entrust their papers to a new venue, the program committee members who agree to read and review the submissions, the attendees who travel long distances to listen and participate, the individuals who invest time planning and running the conference, and, finally, the organizations that sponsor it. In all of these aspects, NSDI's maiden voyage has been a charmed one.

We received 118 submissions on a wide range of topics with an unusually high level of quality and creativity. Each of these papers received written reviews from at least three program committee members, and the top 54 submissions were selected for two additional reviews---463 written reviews in all. The papers were vigorously debated for over eight hours at the program committee meeting in San Diego, and ultimately 27 of the highest-quality works were selected for publication. Each paper was assigned an individual shepherd from the program committee and went through multiple revisions before inclusion in these proceedings. The resulting program is strong by any measure and covers a wide variety of current topics, ranging from sensors networks and model checking to secure routing and peer-to-peer systems.

While a conference is frequently measured by the quality of the papers in the proceedings, in truth there is much more that goes on behind the scenes to ensure success.

First and foremost, NSDI has benefited from the hardworking USENIX staff, who have an uncanny knack for putting a conference together in spite of the obvious limitations and frailties of the conference chairs. In particular, we offer our gratitude to Ellie Young, Jane-Ellen Long, and Anne Dickison, who were a joy to work with and were consummate professionals. We would also like to thank SIGOPS Chair Keith Marzullo and SIGCOMM Chair Jennifer Rexford for lending the support of their organizations. Finally, Ratul Mahajan, David Wetherall, and Tom Anderson deserve special credit for running the NSDI "Shadow PC" experiment, which provided graduate students with early exposure to the process of program selection.

In conclusion, we believe that this first NSDI features an outstanding program, constituting some of the best systems research being done today. We look forward to seeing you all there!

Robert Morris, MIT
Stefan Savage, University of California, San Diego
Program Chairs

The Emergence of Networking Abstractions and Techniques in TinyOS

Philip Levis[†], Sam Madden[*‡], David Gay[‡], Joseph Polastre[†], Robert Szewczyk[†], Alec Woo[†], Eric Brewer[†]and David Culler[†]

[†]EECS Department
University of California, Berkeley
Berkeley, California 94720

[‡]Intel Research Berkeley
2150 Shattuck Avenue
Berkeley, California 94704

[*]CSAIL
MIT
Cambridge, MA 02139

Abstract

The constraints of sensor networks, an emerging area of network research, require new approaches in system design. We study the evolution of abstractions and techniques in TinyOS, a popular sensor network operating system. Examining CVS repositories of several research institutions that use TinyOS, we trace three areas of development: single-hop networking, multi-hop networking, and network services. We note common techniques and draw conclusions on the emerging abstractions as well as the novel constraints that have shaped them.

1. INTRODUCTION

Networked systems of small, often battery-powered embedded computers, referred to as EmNets in a recent NRC report [17], are touted as a revolution in Information Technology with "the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways." They are also seen as requiring a dramatically new approach to network system design:

> "EmNets are more than simply the next step in the evolution of the personal computer or the Internet. Building on developments in both areas, EmNets will also be operating under a set of constraints that will demand more than merely incremental improvements to more traditional networking and information technology." [17]

Networking issues are at the core of the design of embedded sensor networks because radio communication – listening, receiving, and transmitting – dominates the energy budget, defining the lifetime of a deployed system.

Much of the research in this area has been based on the TinyOS operating system, created at UC Berkeley [23], but now a public Sourceforge project including many other groups. In the three years since TinyOS's introduction, it has been used by many research groups to address various aspects of EmNet design. Most of this work is available in the open, making it possible to examine the new system structures that have emerged and the common techniques that they exploit. Are they, in fact, substantially different from general purpose systems?

To begin to answer this question, this paper draws on our own experience building TinyOS, TinyDB and other applications as well as a study of development represented by the CVS trees of several other research groups (CMU, UCLA, USC, UIUC, UVA, and Vanderbilt). An appendix contains the list of the sources used to collect this data. We look in particular at code evolution through several generations of hardware platforms, OS releases, and applications. Our primary focus is on networking abstractions as they appear in applications with particular sensing and system management structures. We find that abstractions fall into four categories:

- Certain abstractions appear to be *general*: they are widely used, and TinyOS provides both the mechanism and policy to support them. The use of the active messages networking abstraction [6], multihop broadcast, and tree-based routing are prime examples.
- Others appear to be *specialized*: they are also widely used, but TinyOS provides only mechanisms. Each application includes code to dictate a specific policy. A typical example is power management, where the application decides when to power down particular subsystems.
- Still others are *in flux*, without any consensus on the abstraction. Instead, these abstractions are often implemented as part of an application. For example, epidemic protocols are often used, but the TinyOS community has not developed a common interface or implementation.
- Finally, a set of abstractions are conspicuous by their *absence*, in that they appear widely in the literature, but are scarcely used in the applications we find in our sample set.

Abstractions have moved among these classes, as developers refined them; our classification is merely an observation of the current state of development, to which we look for insight. For example, initially tree-based routing was built into each application. It has since emerged as a general abstraction, with several implementations of a common interface used by multiple applications.

We also identify four system design techniques used in these abstractions that are comparatively uncommon or unimportant in more general systems. First, **cross-layer control**, where two system levels interact in a mutual relationship, is quite common. This reflects the hardware constrained and application specific nature of EmNets, but also the use of a 'design to suit' framework, rather than a strict set of predefined interfaces. Second, there are many instances of rate-matched data pumps, either of network or sensor data; the comparatively small storage on these devices often requires a **static resource allocation** discipline, where queue sizes are carefully considered and allocated at compile-time. Finally, **snooping** and **scheduled communication** pose a design tension. Snooping depends on frequently listening to the radio to collect as much data as possible, while scheduled communication avoids the energy cost of listening by coordinating the schedules of senders and receivers.

Section 2 provides a brief background on TinyOS, major hardware platforms, and three example applications. In Sections 3, 4, and 5, respectively, we examine single-hop (data link) communication, multi-hop (network) communication, and system services of particular importance to networking (time synchronization and power management). For each, we examine systems that have been built by the larger TinyOS community to identify networking abstractions and how they evolved. In Section 6 we survey the identified abstractions and discuss common system design techniques and trends.

2. TinyOS

As background, we review the basic design of TinyOS and summarize the variety of hardware platforms and applications that have driven TinyOS development, focusing on the requirements dictated by each.

2.1 TinyOS Design

TinyOS focuses on three high-level goals in sensor network system architectures [23]:

- Take account of current and likely future designs for sensor networks and sensor network nodes.
- Allow diverse implementations of both operating system services and applications, in varying mixes of hardware (in different mote generations) and software.
- Address the specific and unusual challenges of sensor networks: limited resources, concurrency-intensive operation, a need for robustness, and application-specific requirements.

```
interface StdControl { // booting and power management
  command result_t init();
  command result_t start();
  command result_t stop();
}

interface ADC { // data collection
  command result_t getData();
  command result_t getContinuousData();
  event result_t dataReady(uint16_t data);
}

interface SendMsg { // single-hop networking
  command result_t send(uint16_t addr, uint8_t len,
        TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg, result_t
        success);
}

interface ReceiveMsg { // single-hop networking
  event TOS_MsgPtr receive(TOS_MsgPtr m);
}
```

Figure 1: Common TinyOS Interfaces. *ADC and SendMsg are split-phase operations, hence the combination of commands and events.*

To achieve these goals, TinyOS provides an efficient framework for modularity and a resource-constrained, event-driven concurrency model. The modularity framework allows the OS to adapt to hardware diversity while still allowing applications to reuse common software services and abstractions.

The need to handle high concurrency comes from the observation that sensor nodes predominantly process multiple information flows on the fly, as opposed to performing heavy computation. Nodes must simultaneously execute several operations at once, but have limited storage. Combined with the need for sensor nodes to be robust, an event-driven concurrency model best suits this class of system. Therefore, all of the abstractions and techniques we explore are cast in an event-driven model.

Program execution in TinyOS is rooted in *hardware events* and *tasks*. Hardware events are interrupts, caused by a timer, sensor, or communication device. Tasks are a form of deferred procedure call that allows a hardware event or task to postpone processing. Tasks are *posted* to a queue. As tasks are processed, interrupts can trigger hardware events that preempt tasks. When the task queue is empty, the system goes into a sleep state until the next interrupt. If this interrupt queues a task, TinyOS pulls it off the queue and runs it. If not, it returns to sleep. Tasks are atomic with respect to each other.

System modularity is based on a component model. A named component encapsulates some fixed-size state and a number of tasks. Components interact with each other strictly via function call *interfaces*, which are related groups of *commands* and *events*. The set of interfaces a component uses and provides define its external namespace. Commands typically represent requests to initiate some action; events represent the completion of a request or something triggered by the environment, e.g., message reception. Both explicitly return error conditions, such as the inability to service the request. A specific set of events are bound to hardware interrupts. A programmer assembles an application by specifying a set of components and "wiring" their interfaces together.

The TinyOS concurrency model intentionally does not support blocking or spin loops. As a result, many operations are *split-phase*: the request is a command that completes immediately, and an event signals completion of the requested action. This approach is natural for reactive processing and for interfacing with hardware, but complicates sequencing high-level operations, as a logically blocking sequence must be written in a state-machine style. It allows many concurrent activities to be serviced on a single stack, reflecting the very limited RAM on modern microcontrollers. It also makes handling of error conditions explicit.

TinyOS intentionally defines neither a particular system/user boundary nor a specific set of system services. Instead, it provides a framework for defining such boundaries and allows applications to select services and their implementations. Nonetheless, a common set of services has emerged and is present on most platforms and used by most applications. This include timers, data acquisition, power management, and networking. Some interfaces supporting these services are shown in Figure 1. This general decomposition is not unique to TinyOS; it can be found in other sensor network platforms, such as SensorSim [36], MANTIS [2], and EmStar [15]. It covers the basic requirements of a sensor network: using little power, periodically collect data, perform some simple processing on it, and, if needed, pass it to a neighboring node.

2.2 Hardware Platforms

There has been a dramatic evolution in hardware platforms since TinyOS was first designed. Table 1 summarizes the generations of the processing board of Berkeley "motes" and similar designs from other groups. Although the Berkeley mote's microcontrollers are drawn from the same family (Atmel AVR), the radios, their interfaces, and the chip-to-chip interconnects differ substantially. Other hardware platforms that support TinyOS by Intel [26] and the ETH [28] use different processors (ARM) and/or different radios. Several companies have developed other variants. Highly integrated experimental devices with a processor and radio integrated on a single tiny die also exist [22]. We examine the impact these hardware variations have on networking in greater detail in Section 3.

While it is reasonable to expect that the capability of a device with a given size and power limit will improve with time, but the rate of improvement is unlikely to mirror that of desktop systems and there will be similar pressures to reduce the size and power consumption for a given capability. The balance of processing, memory, and communication for the designs in Table 1 is very far from the 1 MIPS:1 MB:1 mbps rule of thumb. Small, simple processors are fast, but memory accounts for most of the microcontroller chip and most of the standby power consumption. The radio consumes the majority of the active power.

2.3 Applications

Sensor network development has been very application driven. TinyOS applications have evolved from simple sense-and-route demos to a variety of complete applications, some of which have had long term deployments. We present three examples that illustrate a range of sensor network abstractions:

Habitat Monitoring (TinyDB): patches of nodes gather sensor data for several months from areas of interest to natural scientists or other environmental observers. Example deployments include James Reserve [32], Great Duck Island [31] and a vineyard in British Columbia [44]. We use TinyDB [30] as a representative habitat monitoring application. It allows a user to collect data from a sensor network using a flexible database-like query language. Nodes cooperatively process queries to collect and extract the requested data. TinyDB represents a class of stationary networks that monitor the encompassing space.

Shooter Localization: a small subset of a sensor network localizes the origin of a bullet in an urban setting [27]. Individual nodes detect projectile shockwaves (with latency in the tens of microseconds) and the sound of the weapon firing, and use the time between the two to estimate distance. Actual shooter location is computed centrally. Shooter localization represents a class of stationary networks that monitor objects or phenomena within a space.

Pursuer-Evader: a large network tracks the movement of one or more evader robots using magnetic or other field readings. The network routes this information to a pursuing robot using geographic or landmark routing [9]. Pursuer-evader represents a closed-loop network with stationary and mobile nodes, and was developed by several groups as part of a demo for the EmNet-related DARPA program, NEST [40].

These three applications drive three different areas of TinyOS development: Habitat Monitoring needs to keep energy consumption low, so that multi-month de-

Mote	WeC	rene	dot	mica	mica2	mica2 dot	iMote [26]	btNode[25]
Released	1999	2000	2001	2002	2003	2003	2003	2003
Processor	4 MHz				7 MHz	4 MHz	12 Mhz	7 Mhz
Flash (code, kB)	8	8	16	128	128	128	512	128
RAM (kB)	0.5	0.5	1	4	4	4	64	4
Radio (kBaud)	10	10	10	40	40	40	460	460
Radio Type	RFM				ChipCon	ChipCon	Zeevo BT	Ericson BT
μcontroller	Atmel						ARM	Atmel
Expandable	no	yes	no	yes	yes	yes	yes	yes

Table 1: Hardware Platform Evolution

ployments are possible; Shooter Localization requires a high sample rate and fine-grain time synchronization; Pursuer-Evader requires mote localization and more advanced routing (to the mobile pursuer).

3. SINGLE HOP COMMUNICATION

Active messages [6] has remained the basic networking primitive of TinyOS. Active messages is a simple message-based networking abstraction where messages include an identifier that specifies an action to be executed upon message reception. Although the abstraction has changed little, its implementation and features have changed substantially, due to changing hardware platforms and emerging needs. There have been three major networking stacks from UC Berkeley, one for each of the primary platforms: rene, mica, and mica2. The primary differences between these stacks can be traced to shifts in the network hardware/software boundary.

The core challenge has been to meet the hard real time requirements of networking hardware. The radio timing requirements have influenced and restricted the use of tasks not only in the radio stack, but throughout TinyOS. Reliable radio reception requires high-frequency low-jitter channel sampling. This is simplified by raising the hardware/software boundary. However, raising this interface is not without cost: useful features, such as fine-grained power management, time-stamping, selective back-off, and link-level packet acknowledgment, become more expensive and complex to provide on more sophisticated radio interfaces. Leopold et al. [25] observe similar issues with the Bluetooth interface:"the arguments mentioned above [high energy utilization, lack of timestamps, no access to connectivity data] disqualify Bluetooth as a first choice for sensor nodes."

3.1 Communication Interfaces

In most cases, a macro-component called GenericComm encapsulates the TinyOS network stack. GenericComm provides the active message communication interfaces (SendMsg and ReceiveMsg, as shown in Figure 1), which support single hop unicast and broadcast communication. Message buffer structures have a fixed size. CSMA (carrier-sense, multiple access) is the default media access. Applications and the network stack exchange message buffers through pointers: senders provide their own storage for messages (there is no send buffer pool). Successful calls to send implicitly yield the provided buffer to GenericComm, which returns it to the sender when signaling the sendDone event.

GenericComm provides limited receive buffering. When a component receives a message, it must return a buffer to the radio stack. The stack uses this buffer to hold the next message as it arrives. Typically, a component consumes and returns the buffer it was passed; however, if it wishes to store this buffer for later use, it may instead return a pointer to another free buffer. If it has exhausted all its buffering, the component must determine what should be dropped. The stated goal is to keep the system responsive and ensure that there is always a free buffer for the radio stack to use. Components that need to accumulate several packets before processing them must allocate buffers statically and locally [11], rather than relying on the network stack to provide arbitrary buffering.

Alternative interfaces for single hop communication exist. For instance, S-MAC [45] is an RTS/CTS-based networking stack that supports fragmentation at the data link level. This allows S-MAC to support sending messages larger than the link MTU (maximum transmission unit). It schedules RTS/CTS exchanges and turns radios off to reduce listening costs transparently. S-MAC preserves the split-phase nature and zero-copy semantics similar to SendMsg in its byte stream interface and provides an Active Message interface.

Based on the code we reviewed in CVS, almost all applications developed under TinyOS use active messages. It appears that the availability of large message sizes is not yet a general need: in the few cases where the default message size has been insufficient, applications have increased it modestly (e.g., from 36 to 56 bytes in TinyDB). TinyDiffusion, a naming protocol discussed in Section 4.2, uses S-MAC, but does not send messages larger than the link MTU.

We now examine in detail each of the three implementations of active messages, beginning with the earliest implementation, for the rene mote. We emphasize the differences between successive implementations, noting ways in which the hardware/software boundary changed and which of these ways most influenced the capabilities

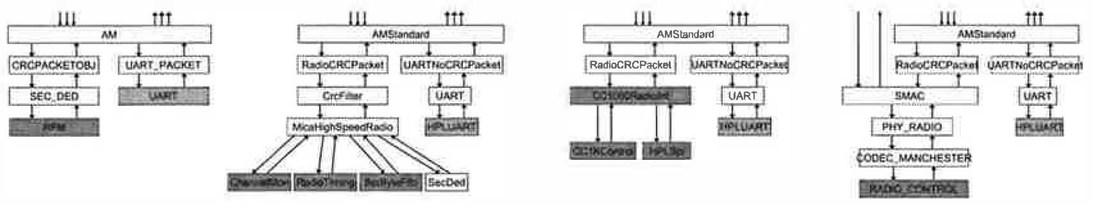


Figure 2: Typical single hop network stacks for three generations of motes. From left to right: rene, `mica`, `mica2`, and S-MAC on `mica` radio stacks. Grey components abstract hardware.

of the system.

3.2 AM Stack Implementations

Figure 2 shows the structure of the various stack implementations. The rene RFM TR1000 radio is extremely simple. The system modulates the channel by writing a DC-balanced bit sequence to the transmit pin at precise times. Similarly, the data is recovered by reading the receive pin at precise times to sample the channel. The RFM component exports this bit-level interface. Bits are received and transmitted in a timer interrupt handler. CSMA media access control is intertwined with data encoding and transmission within the SEC_DED component. CRCPACKETOBJ performs the packet-to-byte translation, with CRC checking, while AM is responsible for application layer dispatch.

The mica hardware uses the same radio as the rene and can use the rene stack unchanged. However, the mica adds hardware edge detection and byte-to-bit conversion on top of the RFM chip. This allows two major advancements: more precise bit timing, and increased data rates (40KBaud over rene's 10KBaud). Instead of a strictly layered approach, the mica stack delegates separate parts of the communication process to different components. It separates out the media access control (ChannelMon), bit-level timing (RadioTiming) and data transfer (SpiByteFifo). MicaHighSpeedRadio signals packet reception events to the active message layer as well as low-jitter timing events at specific points in packet reception or transmission (e.g., when the channel is acquired). Interpositioning is used to provide CRC checks on the packet level (through CRCFilter).

A later revision of the stack introduced synchronous link-layer acknowledgments. The sender and receiver swap roles just as the last data byte of the packet is sent. The receiver transmits an acknowledgment code (a predefined sequence of bits), and the sender reports if it heard the code in its sendDone event. The sender then has a synchronous acknowledgment at the cost of a few byte times and can immediately decide whether to retransmit the buffer or use it for a new message.

Low power listening reduces the cost of idle listening by lowering the radio sample rate when looking for a packet start symbol and turning off the radio between samples. This requires a transmitter to send a longer start symbol. On the RFM TR1000, which can turn on in less than a bit time, low power listening can lead to significant idle listening energy savings at only a small overhead to a transmitter. Once a receiver detects a packet, it keeps its radio on and receives at the full data rate.

The mica2 is based on a Chipcon CC1000 radio, which performs bit synchronization and encoding and exports only a byte-level interface. A single, large component, CC1000RadioInt, replaces the modular architecture used in mica. Initially, the mica2 stack did not provide link-layer acknowledgments, but later revisions added them as well as low power listening. The application-level semantics of CC1000 acknowledgments are the same as the mica. However, the hardware interface requires the implementation to send a tiny ACK data packet instead of an ACK code, and low power listening is not nearly as efficient.

New platforms are emerging that support the IEEE 802.15.4 standard for personal area wireless networks. 802.15.4 radios provide packet level interfaces for sending and receiving, link layer encryption and authentication, link-layer acknowledgments, CRC checking, and early packet rejection. These new standardized radios move most of the functionality of the CC1000 radio stack into hardware. simplifying the TinyOS code.

The S-MAC stack in Figure 2 shows an alternative approach to layering. The complex SMAC component handles backoff, retransmission, RTS/CTS/ACK handshakes, fragmentation, and radio duty cycle control. PHY_RADIO performs CRC checking and byte buffering while CODEC_MANCHESTER runs Manchester coding. RADIO_CONTROL handles the low-level carrier sensing and start symbol detection.

3.3 Analysis

Hardware differences between mote platforms affected software structure and networking capabilities in unforeseen and surprising ways. We discuss four examples: hard real time issues, low power listening, link layer acknowledgments, and low-level hooks.

The concurrency model was intended to allow a collection of components to be replaced by hardware and vice versa. Moving the hardware/software boundary changes the division of work between tasks and hardware events, leading to different maximum critical section and task lengths. The bit-level interface in rene requires the stack to handle interrupts at a high enough rate that the handler cannot encode or decode bytes; encoding/decoding is deferred to tasks. Because it has a small buffer, the encoding layer (SEC_DED) must run a task every byte time. This means no other tasks can run longer than a byte time (1.8ms). A larger encoding/decoding buffer would reduce sensitivity to long-running application tasks.

In mica, the addition of a byte-level hardware abstraction reduced the interrupt rate. This increased the number of cycles available for each event, allowing encoding/decoding to remain in the interrupt handler. Tasks are posted by the packet level component, making the stack more resilient, but not immune, to the effects of long-running application tasks: the maximum task length increases to a packet time, approximately 25 milliseconds.

Both the rene and `mica2`stacks support low-power listening. However, it is much more efficient in the former, because the TR1000 can turn on very quickly, while the CC1000 turns on slowly. This latency determines how quickly the stack can sample the channel, and therefore how much it can reduce the cost of idle listening.

Synchronous link layer acknowledgments present another timing issue. In the mica stack, they depend on being able to quickly transition (in 12 μs, where a raw bit time is 25μs) the RFM radio between send and receive mode while maintaining synchronization to within a bit time. The Chipcon radio transitions much more slowly (250 μs, where a raw bit time is 26μs) and self-synchronizes at byte boundaries, precluding this technique. For this reason, mica2 acknowledgments require a complete packet transmission. This is a well-known and fundamental tension: moving the hardware boundary further up the stack improves performance, but also disallows certain capabilities enabled by low-level hardware access. The important parameter in this case was transition time, as opposed to more traditional measures of radio performance such as throughput or sensitivity.

An additional feature that emerges is low-level hooks that, following the technique of cross-layer control, allow higher level components to monitor radio events with high precision. We examine one common use of these hooks in Section 5.2, when we discuss time synchronization. They are also commonly used in localization systems based on time of flight measurements [41].

In summary, we observed the emergence of the AM abstraction as general and widely used throughout TinyOS. Lower levels of the radio stack, however, are implementation-specific and tied to a particular hardware platform. The development of the various mote platforms shows that several hardware characteristics enable important features and influence the ability of the radio stack to coexist with applications and services.

4. MULTI-HOP COMMUNICATION

The literature describes many ad-hoc multi-hop routing algorithms, where network routes are discovered through a self-organizing process [9, 13, 37, 38, 43]. Similarly, varieties of multi-hop routing components are among the most diverse and numerous implementations contributed to the TinyOS repository. We broadly divide these components into three classes: *tree-based collection*, where nodes route or aggregate data to an endpoint, *intra-network routing* where data is transfered between in-network end-points, and *dissemination*, where data is propagated to entire regions. Habitat monitoring primarily uses tree-based collection, whereas Pursuer-evader routes between mobile in-network regions. Essentially all applications use some form of broadcast or dissemination to convey commands, reconfigure, or control in-network processing. Examining several multi-hop implementations, we see the emergence of two common abstractions: (i) a neighborhood discovery and link quality estimation service and (ii) an augmented version of the AM interface that supports packet encapsulation and monitored forwarding. We summarize these classes and common abstractions, noting several key points in the evolution of routing as implemented in TinyOS.

4.1 Tree-Based Routing

Tree-based routing is primary based on two pieces of information: a parent node identifier, and a *hop-count* or depth from the tree root, i.e., the parent's hop-count plus one. A routing tree is built via local broadcast from the root followed by selective retransmission from its descendents. A node routes a packet by transmitting it with the parent as the designated recipient. The parent does the same to its parent, until the packet reaches the root of the tree. The key design issues are how the routing tree is discovered and maintained, as well as how the forwarding is performed. We examine the historical improvements of tree formation over five successive protocols (AMROUTE, BLess, Surge, mh6, and MultiHopRouter).

The early AMROUTE builds a tree using periodic beacon floods from the root, keeping no route history; it only maintains a single, current parent, the first receipt of the recent beacon. BLess, in contrast, uses no flooding. Instead, each node listens to routed data packets and greedily selects a parent from overheard transmissions. The root node seeds the tree by periodically broadcasting BLess packets, but these broadcasts are not retransmitted. BLess maintains a table of candidate parents and uses the parent with the lowest hop-count.

Surge also uses beaconless tree formation and maintains a table of candidate parents, but selects its parent based on a combination of link quality (packet success rate) and hop count. mh6 [43] and its re-implementation in MultiHopRouter extend this approach by filtering the neighbor set (based on link quality and other factors) and choosing the parent based on an estimated cost to the root through each neighbor. Each node computes quality estimates on incoming links and periodically transmits these, along with its cost estimate to the root, allowing neighbors to determine outgoing link quality and total path estimates. Parent selection algorithms try to minimize either end-to-end packet loss or, with link-level acknowledgments, total expected transmissions, including retransmissions. Surge and MultiHopRouter provide support for retransmission and output queuing. Similar techniques for asymmetric link rejection and neighborhood management have recently been proposed for ad-hoc routing in 802.11 networks [13].

One reason why many implementations of tree-based protocols exist is that it is straightforward to construct a basic tree-based topology and forwarding mechanism.application. However, substantial care is required to construct and maintain a stable topology that can provide good connectivity in a diverse radio environment for weeks or months. The introduction of parameterized interfaces in NesC, combined with the development of more mature routing protocols made it possible to build a robust, encapsulated routing layer that could be easily reused in many applications.

4.2 Intra-network Routing

Several established ad hoc protocols, such as DSDV, AODV, and Directed Diffusion, have been implemented for TinyOS in various reduced forms. DSDV and AODV are designed for unicast routing to specific endpoints. In the TinyOS version implemented by Intel Oregon [44], the basic algorithms for route discovery and maintenance are similar to their IP counterparts, but the final results are not. Instead, they maintain a single route, much as tree-based protocols do. TinyDiffusion builds, then prunes, a routing tree based at a particular requesting source. [19]

A number of protocols structure network names to assist routing. A fully featured implementation of GPSR [24], which uses a node's geographic location as its name, has been completed by University of Southern California. It supports both greedy geographic routing and perimeter routing to recover from local failures. Pseudo-geographic routing, in which each node is assigned a vector of hop-count distances from several beacons instead of coordinates in a Cartesian plane, has been explored as well. [38] Intra-network routing, the mainstay of Internet usage, is uncommon in TinyOS applications. One exception is Pursuer-Evader, which provides mobile-to-mobile routing within a single routing tree rooted at a landmark. The destination reinforces a path from the landmark for downward routing.

4.3 Broadcast and Epidemic Protocols

Many applications need to reliably disseminate a data item to every node in a network. For example, in TinyDB new queries must be installed, while in Pursuer-Evader the application can be reconfigured in-situ (e.g., to change filter settings or radio transmit power). Reliable network-wide dissemination can also be used to distribute new versions of TinyOS programs.

In practice, reliable data dissemination has two principal implementations. The first, a simple flooding protocol, is common, easily implemented, and often tightly integrated in an application; it appears in Pursuer-Evader as well as others. In this implementation, the source (usually a base station) generates a data packet, and each node that hears it forwards it once. Experience has shown that a single flood often reaches most nodes very quickly, but collisions and lossy links lead to several motes not hearing the data. Typically, the source repeats the flood several times, until every node receives it. Determining when to stop requires either visual inspection or routing data out of the network to the base station. Pursuer-Evader uses a delayed retransmission strategy, which greatly reduces collisions and improves coverage.

The second approach is to use an *epidemic* algorithm. Instead of a single flooding event, nodes periodically exchange information to know when to propagate data [14]. For example, the Maté virtual machine uses a purely epidemic algorithm to disseminate new code [29]. In contrast to a flood, an epidemic only transmits when needed. Local suppression mechanisms can reduce redundant transmissions, saving energy. By ensuring reliable delivery to every connected node, an epidemic approach is robust to transient disconnections and can propagate to new nodes added to a network.

TinyDB uses a hybrid approach to install and stop queries. It first floods new queries into the network; this reaches most nodes very quickly. The network then uses an epidemic approach to reach the few remaining nodes that missed the flood. As every data message contains a query ID, nodes can detect inconsistencies by snooping on local data traffic.

4.4 Common Multi-Hop Developments

We observe a number of common developments that have occurred in several multi-hop networking implementations for TinyOS. First, most of the current multi-hop routing implementations – MultiHopRouter, TinyDiffusion, GPSR, BVR – discover and manage a list of neighboring nodes for possible routes. They use this information when initially constructing routes and to adapt to connectivity changes, including node appearance and

```
interface Send {
  command result_t send(TOS_MsgPtr msg, uint16_t
       length);
  command void* getBuffer(TOS_MsgPtr msg, uint16_t*
       length);
  event result_t sendDone(TOS_MsgPtr msg, result_t
       success);
}
interface Intercept {
  event result_t intercept(TOS_MsgPtr msg,
                    void* payload,
                    uint16_t payloadLen);
}
```

Figure 3: The Send and Intercept Interfaces for Routing.

disappearance. Typical information appearing in neighborhood tables includes node addresses, link quality estimates, and routing metadata, such as hop-count in routing-tree protocols. Note that this table contains both link state (link quality) and routing layer (hop-count) information. With limited memory, the table is constrained to a limited number of entries. Routing components utilize link information in route selection, while link components utilize routing information in table management. Aspects of both are conveyed in route update messages.

Recent experiments have established that sensor networks experience lossy and asymmetric links whose quality changes over time [7, 42]. Traditional protocols that assume bi-directional connectivity are likely to fail in such an environment; many early multi-hop routing protocols in TinyOS correspondingly suffer from poor end-to-end packet delivery [20]. To remedy this, recent routing layers in TinyOS, such as MultiHopRouter, BVR, TinyDiffusion and the TinyOS DSDV implementation include the notion of a *link quality estimator* that identifies a set of bi-directional, high-quality links that network, transport, and application layers can rely upon.

A second common development in multi-hop routing is that routing layer implementations (e.g., DSDV, MultiHopRouter) have begun to use the Send and Intercept interfaces, shown in Figure 3. The getBuffer command in the Send interface allows the routing layer to control the offset of the application payload in a message buffer, which is useful for packet encapsulation. A routing layer signals the Intercept event when it receives a packet to forward. An application can suppress forwarding by returning a certain result code. This allows application such as TinyDB to locally aggregate data. These interfaces are examples of cross-layer control, and enhance the AM abstraction with support for multi-hop communication by interposing new interfaces. Similar interfaces would support broadcast with processing at each hop.

A third development is augmenting low-level network abstractions to include interfaces for promiscuous communication, where the network stack can pass non-locally addressed packets up to a higher level component. Some link estimation and neighbor table management modules rely on this to learn about nearby nodes. TinyDB uses it for application-specific network optimizations. This prevalent use of snooping indicates that it is a technique of general utility in sensor networks.

Finally, a fourth development that has recently emerged in multi-hop protocols is the addition of a send queue. Initial implementations, such as BLess, have a single packet buffer, and drop outgoing packets when the underlying communication components are busy transmitting. Surge, mh6, and MultiHopRouter all add outgoing queues, but have different queuing policies. For example, mh6 maintains separate forwarding and originating queues, giving priority to originating, while MultiHopRouter gives priority to forwarding. No implementations that we found include a receive queue beyond the simple buffering provided by AM.

4.5 Observations

Except for GPSR, all multi-hop protocol implementations in TinyOS we could find are built on top of the AM abstraction. The stability and wide use of the AM interface suggest that protocol developers are satisfied with the simple dispatch function it provides. It is possible that this stability reflects a desire not to modify the lower level portions of the network stack, which tend to be complex. However, the untyped packet interface below AM is similarly isolated from this complexity. It is worth noting that TinyDiffusion, which is built on top of S-MAC, currently uses S-MAC's AM interface rather than an interface which S-MAC provides that includes features like fragmentation.

When a message exceeds the length of an AM packet, application level framing is used [10]. For example, the TinyDB application partitions both queries and query results into logical units that have meaning on their own, (e.g., a field in a query result.) If a single fragment gets lost, the application can still use the other fragments.

Applications developed on TinyOS have predominantly used tree-based routing. Some applications, such as pursuer-evader, use intra-network routing, but implementations tend to be single destination and fairly simple. The literature, however, has proposed many intra-network routing algorithms. There are several potential explanations for this difference in usage. Complex routing algorithms may have difficulty scaling down to resource constrained nodes. The implementations may simply be too immature enough to have seen use in released applications. This was the case with initial implementations of tree-based routing which were not sufficiently robust to be widely adopted (despite their simplicity). However, it appears that general point-to-point routing is simply less common in applications of embedded networks, which tend to operate in aggregate on information that is distributed throughout the network. In such applications, information tends either to be broadcast out or to flow in a single direction towards a small number of sinks.

Applications are increasingly using reliable dissemination for programming or reconfiguration. For small networks, simple floods are sufficient. As sensor networks are deployed at larger scales, the need to respond to unforeseen system interactions increases, as does the benefit of an epidemic-based solution. However, unlike prior IP-based epidemics, sensor nets can take advantage of geographic proximity and spatial redundancy.

Based on the observations in this section, we note several progressions in the development of multi-hop networking in TinyOS. First, the evolution of a neighborhood management table with the ability to reject asymmetric links and select low-loss routes has finally resulted in a widely used tree-based routing component – MultiHopRouter. Second, the use of snooping (as opposed to broadcast floods) to gather neighborhood information and construct initial routes has become standard practice. Finally, the appearance of send queues suggests that applications may need to be tolerant of significant transmission delays and that some traditional networking techniques (e.g., differentiated services or load shedding) are applicable in this domain.

5. NETWORK SERVICES

A number of abstractions support efficient, low-power networking in TinyOS. In this section, we focus on two prominent examples, power management and time synchronization.

5.1 Power Management

TinyOS manages power management through the interaction of three elements. First, each service can be stopped through a call to its StdControl.stop command (see Figure 1); components in charge of hardware peripherals can then switch them to a low-power state. Second, the HPLPowerManagement component puts the processor into the lowest-power mode compatible with the current hardware state, which it discovers by examining the the processor's I/O pins and control registers. Third, the TinyOS timer service can function with the processor mostly in the extremely-low-power *power-save* mode.

TinyDB uses these features to support sensor network deployments that last for months. In this context, idle listening dominates energy consumption. As discussed in Section 3.2, low-power listening reduces the cost of idle listening by increasing the cost of transmission. However, instead of low power listening, TinyDB uses communication scheduling. Using coarse-grained (millisecond) time synchronization, TinyDB motes coordinate to all turn on at the same time, sample data, forward it to the query root, and return to sleep.

Figure 4 illustrates TinyDB's power management approach. At the end of a round of data collection, each mote calls StdControl.stop to stop both the onboard sensor hardware (IntersemaPressure) and the radio (CC1000M, UARTM). After the next timer event, HPLPowerManagement puts the processor into power-save mode (via adjustPower). At the start of the next data collection round, the timer wakes the mote up, and StdControl.start is called to restart the sensors and radio. This approach to communication scheduling requires time synchronization when used in conjunction with multi-hop routing, discussed below.

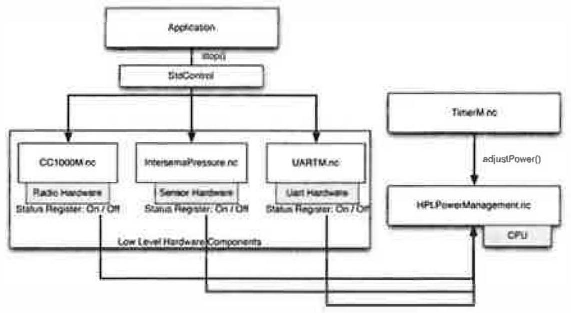


Figure 4: The TinyDB power management API. The application calls StdControl.stop to halt the low-level hardware. HPLPowerManagement.nc sees changes to the hardware status registers, which causes it to put the CPU into a low-power sleep state.

Power management illustrates cross-layer control at a very low-level: HPLPowerManagement goes directly to the hardware to determine when the processor can be switched into various low-power modes (e.g., idle, power-down, power-save, standby, extended standby). Correspondingly, power management is an abstraction that must inherently be specialized: effective power management without application input is not possible. For example, by supplying a small bit of application information, TinyDB allows TinyOS to spend most of its time in a very low power mode. Approaches such as S-MAC take a static approach to this communication scheduling, always waking at a certain fixed interval. TinyDB, however, allows this scheduling to dynamically change in a fine grained manner with regards to application needs (e.g. query sampling rates), conserving more energy.

5.2 Time Synchronization

Another service that many network-centric applications need is time-synchronization. Such a service is useful in several scenarios. For example, sensor fusion applications that combine a set of coincident readings from different locations, such as shooter localization, need to establish the temporal consistency of data. TDMA-style media access protocols need fine-grained time synchronization for slot coordination, and power-aware approaches to communication scheduling (discussed above) require senders and receivers to agree when their radios will be on.

Several groups, including UCLA[1], Vanderbilt[2], and UC Berkeley[3], have implemented time synchronization. GenericComm provides a hook for modifying messages just as it transmits the first data bit, after media access. All of these implementations work similarly: they use the hook to place a time stamp in the packet. This allows very precise time-synchronization that might otherwise not be possible [16]. This is another example of cross-layer control in TinyOS. The Vanderbilt implementation models possible delays, which allows it to obtain slightly better accuracy than the other approaches; all three report sub-millisecond accuracy.

Initial efforts to develop a general purpose, low-level time synchronization component were unsuccessful. A number of subtle and bad interactions observed with some higher level applications, such that timer events are missed or software components hang in inconsistent states. Applications were fragile to time-critical intervals suddenly becoming slightly shorter or longer.

Instead, the current approach taken by TinyOS is to provide the mechanism to get and set the current system time (and time stamp messages at a low level), but to depend on applications to choose *when* to invoke synchronization. For example, in the TinyDB application, when a node hears a time-stamped message from a parent in the routing tree, it adjusts its clock so that it will start the next communication period at exactly the same time as its parent. It does this by changing the duration of the sleep period between communication intervals, rather than changing how long the sensor is awake, since cutting the waking period short could cause critical services, such as data acquisition, to fail.

As with power management, it appears that time synchronization is emerging in TinyOS as a specialized abstraction, with mechanism provided by the operating system and policy by the application. Applications have a varying set of time synchronization requirements, so incorporating their behavior in the service is beneficial. Gradual time shifting is suitable in some situations, while others require sudden shifts to the correct time.

The fact that applications fail when time-synchronization changes the underlying clock has been observed before; systems such as NTP [33] work around this problem by slowly adjusting the clock rate to synchronize it with neighbors. The NTP approach has the potential to introduce errors in an environment as time-sensitive as TinyOS, where a timer that fires even a few milliseconds earlier than expected can cause radio or sensor data to be lost.

[1]In `tinyos-1.x/contrib/Timesync-NESL-UCLA/`

[2]In `tinyos-1.x/contrib/vu/`

[3]In `tinyos-1.x/beta/TimeSync/`

Some applications may prefer to focus on robustness rather than maximizing the precision of time synchronization, as most algorithms proposed in the literature [16, 18] and included in the TinyOS repository strive to do. For example, TinyDB only requires time synchronization to millisecond fidelity, but also requires a rapid settling time. A general abstraction that fit all possible needs would do much more work than TinyDB needs. Simple, specialized abstractions are a natural way to address these types of services.

6. DISCUSSION

We focus now on what can be distilled from this broad study of TinyOS's networking software. We first revisit the four classes of abstractions that have emerged, discussing the members of each class. We also observe that there are design techniques commonly used in TinyOS, but which are not common to more general purpose systems. We then contrast the observed design goals of sensor networks and Internet-based systems.

6.1 Abstractions

We place the observed abstractions into one of four classes: general, specialized, in-flux, or absent. In this last class we note two abstractions that one might expect, based on literature and other networking systems, but were absent in the sources we examined.

6.1.1 General Abstractions

We noted several *general* networking abstractions, i.e, widely used and support by both TinyOS mechanisms and policies. The AM abstraction (Section 3.2) has remained stable since the earliest TinyOS work [23], and most network applications use it either directly or indirectly. This is not entirely surprising, as communication is the core service offered by TinyOS.

Another reason for the stability of AM is that the interface is very simple and lightweight. Other abstractions can easily provide it, in order to be compatible with pre-existing code. For instance, S-MAC provides an implementation of AM on top of its interface, allowing its use by existing programs without modification, despite that fact that it also provides another messaging interface.

A second set of general abstractions have emerged for tree-based routing, particularly the Send and Intercept interfaces shown in Figure 3. Implementations from Berkeley (Route) and Intel-Portland (HSN, AODV, DSDV) use this interface, and major applications, including TinyDB, have been reimplemented to make use of it. From a networking perspective, this is an important development, as it allows applications make use of a variety of different multi-hop implementations without source code modification.

6.1.2 Specialized Abstractions

Specialized abstractions, i.e., those where TinyOS provides mechanisms and applications provide policies, have appeared for both power-management and time-synchronization. In many cases, it is possible to conceive of completely general versions of these abstractions, and general purpose operating systems often provide some version thereof. However, as with the effort to build a general version of time synchronization in TinyOS, general abstractions of some services are very hard to get right. This is because the requirements of applications vary dramatically: some applications and services need time synchronization that is accurate to within a few milliseconds with a small set of other nodes (e.g., TDMA), while others depend on a globally synchronized clock that is much less accurate (e.g., TinyDB). Many sensor network applications have long sleep periods, which provide natural, application-specific points for clock adjustment.

6.1.3 In-Flux Abstractions

A third class of abstractions we consider to be *in-flux*, commonly found but changing between applications and hardware versions, often in conflicting fashion.

One such abstraction is epidemic propagation. Both TinyDB and Maté use this technique to reliably disseminate code (capsules or queries) through a network; we believe that this abstraction is important for sensor networks and expect that it will see further use, e.g., for multi-hop network reprogramming. Many applications propagate commands that need to be received everywhere through naïve flooding. However, no established interface has emerged for this dissemination. This abstraction is a case where the literature [8, 35] provides clear evidence of the shortcomings of flooding and its variants. Based on our observations, however, systems have suffered with these problems instead of adapting and adjusting solutions from prior work to the different constraints wireless sensor networks pose.

An abstraction that is surprisingly in flux is the radio MAC. We find considerable variation in how the channel activity is sensed, the use of control packets (RTS/CTS and ACK) per data packet, backoff, power management, link estimation, queuing, and assumptions about message size and traffic pattern. Additionally, each generation of hardware presented a distinct set of requirements for channel coding, start-symbol detection, alignment, and data transfer. We anticipate that the recent appearance of 802.15.4 radios will stabilize work in this regard; being an accepted IEEE standard gives it legitimacy and wide use. For media access protocols to stabilize they will need to have cross-layer control interfaces, discussed below, so they can be specialized to the particular needs of the application and network layer.

Routing from an arbitrary source to an arbitrary destination appears in only a few applications and involves a small subset of source-destination pairs within the context of substantial dissemination and collection traffic. Currently, it is realized out of the tree-based structures that are used for those other patterns.

6.1.4 Absent Abstractions

Finally, we note a few abstractions that we expected to find in TinyOS based on our reading of the sensor network literature, but that were absent in the code base.

One example is distributed cluster formation, about which there has been a large amount of publication in the ad-hoc and sensor network communities [12, 3, 5]. Instead we find dissemination using simple best-effort broadcasts and collection using continually monitored trees. The prime exception is the Intel DSDV implementation, which builds clusters using an available energy metric. The absence here may stem from the complexity of maintaining 2-hop neighbor lists with low-power radios that are heavily influenced by environmental factors.

Another abstraction missing from TinyOS is incoming (receive) queues. Applications generally handle message reception by accepting a message from the radio stack, processing it, and returning it back to the radio stack. Packets are typically dispatched to components that do a particular, simple operation on the contents. Compared to their communication bandwidth, motes have an abundance of CPU cycles; they can process messages at the rate they receive them.

6.2 Common Techniques

In addition to these abstractions, we observe certain design *techniques* that have been widely and successfully employed in TinyOS. We strive, in particular, to identify common approaches that facilitate program design and engineering or enhance performance. We consider those strategies that are prevalent in the TinyOS codebase, and which the preceding sections suggest will continue to be important in future sensor network software systems.

6.2.1 Communication Scheduling and Snooping

Two conflicting techniques that are widely used in TinyOS, and seem particularly important for sensor networks, are *communication scheduling* and *snooping*. Communication scheduling, as discussed in Section 5.1, refers to disabling the radio except during pre-arranged times when a pair of nodes expect to exchange data. It may also involve frequency or code division. In contrast, snooping refers to receiving packets that might not even be destined for a node, to acquire network neighborhood information or learn about new processing tasks (as with queries in TinyDB or program capsules in Maté); snooping is an essential part of the epidemic algorithms we noted throughout TinyOS.

Snooping tends to reduce the overall communication burden on the network, but requires the node to spend energy listening to its radio and receiving packets – exactly what scheduled communication avoids. While always-on and totally scheduled represent extreme points, we find that applications tend to strike a balance using partially scheduled techniques. For instance, TinyDB leaves the radio on for longer than a single transmission time so that some snooping can occur. Several TDMA schemes are in development [21], but some use the time slots as a heuristic to make collision improbable (using CSMA within slots), instead of assuming the time slot allocation is absolutely collision free. Applications may desire to open up additional 'snooping slots'.

6.2.2 *Cross-Layer Control*

One approach that has emerged as particularly effective in the sensor network domain is the general technique of *cross-layer control*. An example of this is the way in which many TinyOS routing stacks share network neighborhood information between link state and network layers. Exposing state from a lower level (the link state layer) to a higher level (the network layer) avoids duplicating data in multiple layers, conserving RAM. Another example is the provision by the network stack of low-level information, such as received signal strength, to higher layers. Promiscuous communication works similarly: non-locally addressed packets overheard by the network layer are exposed to the application layer to allow it to avoid unnecessary communication.

Cross-layer techniques also work in the opposite direction: higher level components (frequently the application), can use logic in lower components to improve performance. For example, applications write timestamps in time synchronization messages just as the stack sends them using a hook the stack provides. In an extreme case, one radio stack exposes control of MAC layer functionality and parameters.

One reason that these techniques are so effective in sensor networks, and TinyOS in particular, is that each node is generally dedicated to a single task, allowing the application to choose which instance of a particular abstraction (e.g., S-MAC vs AM) it prefers without concern for the operation of other applications. This approach also supports the most important cross-layer technique that has emerged in TinyOS, application control of network services such as time-synchronization and power management. This control is necessary, as the application is the only software component capable of orchestrating the activities of all of the device's subsystems. This is an example of a case where TinyOS has been particularly successful at translating a stated goal of the project into reality; early work states [17]: "Without the traditional layers of abstraction dictating what capabilities are available, it is possible to foresee many novel relationships between the application and the underlying system."

Of course, cross-layer techniques are not new; they have been widely used in the networking and operating systems communities [4, 34]. The resource constrained nature of sensor networks, however, makes these techniques particularly important. Without them, it is unlikely that there would be sufficient RAM or radio bandwidth available to develop the complex systems that have begun to emerge. Moreover, we find that these techniques are used to enhance control, rather than simply to optimize frequent paths.

6.2.3 *Static Resource Allocation*

The third technique apparent in TinyOS is the notion of *static resource allocation*, or allocating buffers for the network, sensors, UART, and other OS services at compile-time. The prevalence of this static allocation in TinyOS is somewhat controversial, as RAM is valuable. To elucidate the importance of this technique, we consider a critical design parameter of any event-driven system: event-arrival rate.

Assuming that processing resources are sufficient, queue size (n) is determined by the maximum event-arrival rate (r), the number of events processed simultaneously (c), and the processing time per event (t). From Little's Law, we readily derive $n = rt+c$. Many TinyOS applications have processing times much smaller than the inverse of the arrival rate, and process one event at a time, which implies $n = 2$ (one buffer for the current object and one for the next arrival). This is the root reasoning behind the buffer-swapping policy of the network stack; i.e. swap these two buffers on every arrival.

This reasoning argues against dynamic buffer allocation, which assumes that another buffer is always available. Since this is not true given the small amount of memory in today's sensor networks, it is much better to statically allocate the right number of buffers, which is the policy generally adopted in the TinyOS community.

This also leads to better composition, since components reserve the amount of memory they need, making total memory requirements checkable at compile-time. In contrast, two components that depend on dynamic buffer allocation may work fine alone, but not as well together as they unpredictably run out of memory. Static allocation can be wasteful when worst case requirements are much greater than the expected case. For example, TinyDB statically allocates 17 separate send buffers, but it is very unlikely that these would all used simultaneously; if the system were so overloaded, not having a buffer to allocate would be a minor issue.

Finally, it should be noted that although we believe the policy of statically allocating memory is the right approach for sensor networks. TinyOS provides limited tools for understanding static memory needs. Improved profilers or simulation tools that could predict such needs, such as recent work on computing stack usage [39], would be of great value to the community.

6.3 EmNets vs. the Internet

RFC 1958 ("Architectural Principles of the Internet") reads:

> "However, in very general terms, the community believes the goal is connectivity, the tool is the Internet Protocol, and the intelligence is end to end rather than hidden in the network ... connectivity is its own reward, and is more valuable than any individual application" [1]

Examined in this light, the networking abstractions that have emerged in sensor networks are the result of more than chance. They differ from traditional Internet abstractions not only because of resource constraints, which might change, but because of a very different set of goals and principles.

End-to-end connectivity is not the primary goal. Unlike the Internet, which is a collection of independent end points that share a common routing infrastructure, sensor networks are homogeneous systems deployed for an application-specific and collaborative purpose. Every node is both a sensor and a router. The relative costs of communication and computation push the architecture from end-to-end logic to in-network processing. Network neighbors are generally physically close; correspondingly, their sensor readings are often related. Additionally, a wireless medium, combined with geographic proximity, correlates network traffic between neighbors.

7. CONCLUSION

In this paper, we were able to classify the most prominent abstractions in TinyOS based on the degree of consensus shown in our community-wide study of the code base. Among the more mature, general abstractions are active messages and tree-based routing. However, most abstractions are still specialized and application specific, in flux, or largely missing despite being in the literature. We observe a trend where application specific abstractions become gradually become more general over time. This is illustrated, for example by the emergence of tree-based routing in TinyOS.

One major reason for the absence of consensus is the unusual degree to which application specialization matters. This appears to be due to a few factors: first, each mote runs only one application at a time, eliminating the need for shared abstractions; second, power management affects all levels of the system and is essentially always application specific; and third, limited resources lead to specialized implementations offering greater efficiency than their generalized counterparts. The last effect is evident in time synchronization, multi-hop routing, and buffer allocation. A fourth factor is that many applications have real-time requirements that mandate precise control over timing throughout the application, reducing the utility of off-the-shelf components.

We also found several techniques that work well in these systems. The two most obvious are cross-layer control and static resource allocation. The former allows better use of resources and more control over timing (e.g. time synchronization), and is well-supported by the "wiring" language in TinyOS, which exposes the layering and encourages interposition and cross-layer thinking. Part of the success of cross-layer control is the support it provides to application specialization. Static resource planning is more robust, more modular, and forces the author to think about the whole program, including all of its resources.

In conclusion, we see that wireless sensor networking is driven by three differentiating factors: power management, limited resources and real-time constraints. These factors have driven the development of the abstractions and techniques seen above. It remains to be seen how lasting and wide-ranging these trends will be.

8. APPENDIX: CODE SOURCES

The following source code repositories were used to collect the data for this paper.

CENS. From `http://cvs.cens.ucla.edu/viewcvs/viewcvs.cgi/tos-contrib/`. Includes RBS [16] time synchronization.

Rutgers. From `http://www.cs.rutgers.edu/dataman/FourierNet/tos10/distro.html`. Includes an ad-hoc positioning system and based on ranging code from Vanderbilt.

sf.net vert. From `http://sourceforge.net/projects/vert/`. Includes an activation tracking demo and "virtual real time" demo from UVA.

TinyOS Contrib. From `http://sourceforge.net/projects/tinyos/` in the `tinyos-1.x/contrib` directory. Includes PRIME, S-MAC, TinyDiff, SensorIB, tinydb, and hsn.

TinyOS CVS and main UC Berkeley repository; includes all UC Berkeley files. From `http://sourceforge.net/projects/tinyos/` in the `nest, tinyos, tinyos-1.x` and `tos` directories. The "TinyDB" application is available at `tinyos-1.x/tos/lib/TinyDB`.

TinyOS Documentation Project. From `http://ttdp.org`.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grant NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069)

9. REFERENCES

[1] RFC 1958: Architectural Principles of the Internet, 1996.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for MultimodAl NeTworks of In-situ Sensors. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 50–59. ACM Press, 2003.

[3] A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *Proceedings of IEEE INFOCOM*, March 2000.

[4] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[5] S. Bandyopadhyay and E. J. Coyle. An energy efficient hierarchical clustering algorithm for wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2003.

[6] P. Buonadonna, J. Hill, and D. Culler. Active message communication for tiny networked sensors. Available from `http://www.cs.berkeley.edu/~jhill/cs294-8/ammote.ps`.

[7] A. Cerpa, N. Busek, and D. Estrin. *SCALE: A tool for Simple Connectivity Assessment in Lossy Environments*, September 2003.

[8] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *Proceedings of ACM MOBICOM*, Rome, Italy, July 2001.

[9] B. Chen and R. Morris. L+: Scalable landmark routing and address lookup for multihop wireless networks. Technical Report 837, MIT LCS, March 2002.

[10] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of SIGCOMM*, pages 200–208. ACM Press, 1990.

[11] D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A network-centric approach to embedded software for tiny devices. *Lecture Notes in Computer Science*, 2001.

[12] B. Das and V. Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *Proceedings of the International Conference on Computing*, 1997.

[13] D. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of ACM MOBICOM*, San Diego, California, Sept. 2003.

[14] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[15] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. Emstar: An environment for developing wireless embedded systems software. Technical Report 0009, CENS, Mar. 2003.

[16] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, USA., Dec. 2002.

[17] D. Estrin et al. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Acedemy Press, Washington, DC, USA, 2001.

[18] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of SenSys*, 2003. To Appear.

[19] D. Ganesan. TinyDiffusion Application Programmer's Interface API 0.1. `http://www.isi.edu/scadds/papers/tinydiffusion-v0.1.pdf`.

[20] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Technical Report 02-0013, UCLA Computer Science Division, Mar. 2002.

[21] L. Gu. PRIME: A Collision-Free MAC Protocol. Draft. Available from `http://www.cs.virginia.edu/~lg6e/MAC.pdf`.

[22] J. Hill. *System Architecture for Wireless Sensor Networks*. PhD thesis, UC Berkeley, May 2003.

[23] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proceedings of ASPLOS*, pages 93–104, Boston, MA, USA, Nov. 2000.

[24] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proceedings of ACM MOBICOM*, pages 243–254, Boston, MA, USA, 2000.

[25] O. Kasten and J. Beutel. BTnode rev2.2. `http://www.inf.ethz.ch/vs/res/proj/smart-its/btnode.html`.

[26] R. Kling. Intel research mote. `http://webs.cs.berkeley.edu/retreat-1-03/slides/imote-nest-q103-03-dist.pdf`.

[27] A. Ledeczi, K. Frampton, G. Simon, and M. Maroti. Shooter localization problem in urban warfare. `http://www.isis.vanderbilt.edu/projects/nest/documentation/Vanderbilt_NEST_Shooter_SanDiego.ppt`.

[28] M. Leopold, M. B. Dydensborg, and P. Bonnet. Bluetooth and sensor networks: A reality check. In *SenSys '03*, Los Angeles, California, Nov. 2003.

[29] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI 2004)*.

[30] S. R. Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, UC Berkeley, Decmeber 2003. `http://www.cs.berkeley.edu/~madden/thesis.pdf`.

[31] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.

[32] Michael Hamilton et al. Habitat sensing array, first year report. `http://cens.ucla.edu/Research/Applications/habitat_sensing.htm`, 2003.

[33] D. L. Mills. Internet time synchronization: The network time protocol. In Z. Yang and T. A. Marsland, editors, *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994.

[34] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the USENIX OSDI 1996*, October 1996.

[35] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[36] S. Park, A. Savvides, and M. B. Srivastava. SensorSim: a simulation framework for sensor networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 104–111. ACM Press, 2000.

[37] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance-vector (aodv) routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

[38] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proceedings of ACM MOBICOM*, September 2003.

[39] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT 2003)*, 2003.

[40] C. Sharp and S. Shaffert. Road to pursuit/evasion on a sensor network. `http://robotics.eecs.berkeley.edu/~cssharp/NEST-demos-overview-Aug2003.ppt`, aug 2003.

[41] K. Whitehouse and D. Culler. Calibration as parameter estimation in sensor networks. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*.

[42] A. Woo and D. Culler. Evaluation of efficient link reliability estimators for low-power wireless networks. Technical Report UCB//CSD-03-1270, U.C. Berkeley Computer Science Division, September 2003.

[43] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges for reliable multihop routing in sensor networks. In *SenSys '03*, Los Angeles, California, Nov. 2003.

[44] M. D. Yarvis, W. S. Conner, L. Krishnamurthy, A. Mainwaring, J. Chhabra, and B. Elliott. Real-World Experiences with an Interactive Ad Hoc Sensor Network. In *International Conference on Parallel Processing Workshops*, 2002.

[45] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of IEEE Infocom 2002*, New York, NY, USA., June 2002.

Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks

Philip Levis†‡, Neil Patel†, David Culler†‡, and Scott Shenker†*
{pal,culler,shenker}@eecs.berkeley.edu, neilp@uclink.berkeley.edu

†EECS Department
University of California, Berkeley
Berkeley, CA 94720

*ICSI
1947 Center Street
Berkeley, CA 94704

‡Intel Research: Berkeley
2150 Shattuck Ave.
Berkeley, CA 94704

ABSTRACT

We present Trickle, an algorithm for propagating and maintaining code updates in wireless sensor networks. Borrowing techniques from the epidemic/gossip, scalable multicast, and wireless broadcast literature, Trickle uses a "polite gossip" policy, where motes periodically broadcast a code summary to local neighbors but stay quiet if they have recently heard a summary identical to theirs. When a mote hears an older summary than its own, it broadcasts an update. Instead of flooding a network with packets, the algorithm controls the send rate so each mote hears a small trickle of packets, just enough to stay up to date. We show that with this simple mechanism, Trickle can scale to thousand-fold changes in network density, propagate new code in the order of seconds, and impose a maintenance cost on the order of a few sends an hour.

1. INTRODUCTION

Composed of large numbers of small, resource constrained computing nodes ("motes"), sensor networks often must operate unattended for months or years. As requirements and environments evolve in lengthy deployments, users need to be able to introduce new code to re-task a network. The scale and embedded nature of these systems – buried in bird burrows or collared on roving herds of zebras for months or years – requires network code propagation. Networking has a tremendous energy cost, however, and defines the system lifetime: laptops can be recharged, but sensor networks die. An effective reprogramming protocol must send few packets.

While code is propagating, a network can be in a useless state due to there being multiple programs running concurrently. Transition time is wasted time, and wasted time is wasted energy. Therefore, an effective reprogramming protocol must also propagate new code quickly.

The cost of transmitting new code must be measured against the duty cycle of an application. For some applications, sending a binary image (tens of kilobytes) can have the same cost as days of operation. Some sensor network applications, such as Tiny Diffusion [8], Maté [13], and TinyDB [18], use concise, high-level virtual code representations to reduce this cost. In these applications, programs are 20-400 bytes long, a handful of packets.

Wireless sensor networks may operate at a scale of hundreds, thousands, or more. Unlike Internet based systems, which represent a wide range of devices linked through a common network protocol, sensor networks are independent, application specific deployments. They exhibit highly transient loss patterns that are susceptible to changes in environmental conditions [22]. Asymmetric links are common, and prior work has shown network behavior to often be worse indoors than out, predominantly due to multi-path effects [23]. Motes come and go, due to temporary disconnections, failure, and network repopulation. As new code must eventually propagate to every mote in a network, but network membership is not static, propagation must be a continuous effort.

Propagating code is costly; learning *when* to propagate code is even more so. Motes must periodically communicate to learn when there is new code. To reduce energy costs, motes can transmit metadata to determine when code is needed. Even for binary images, this periodic metadata exchange overwhelms the cost of transmitting code when it is needed. Sending a full TinyDB binary image ($\approx$ 64 KB) costs approximately the same as transmitting a forty byte metadata summary once a minute for a day. In Maté, Tiny Diffusion, Tiny DB, and similar systems, this tradeoff is even more pronounced: sending a few metadata packets costs the same as sending an entire program. The communication to learn when code is needed overwhelms the cost of actually propagating that code.

The first step towards sensor network reprogramming, then, is an efficient algorithm for determining when motes should propagate code, which can be used to trigger the actual code transfer. Such an algorithm has three needed properties:

Low Maintenance: When a network is in a stable state, metadata exchanges should be infrequent, just enough to ensure that the network has a single program. The transmission rate should be configurable to meet an application energy budget; this can vary from transmitting once a minute to every few hours.

Rapid Propagation: When the network discovers motes that need updates, code must propagate rapidly. Propagation should not take more than a minute or two more than the time required for transmission, even for large networks that are tens of hops across. Code must eventually propagate to every mote.

Scalability: The protocol must maintain its other properties in wide ranges of network density, from motes having a few to hundreds of network neighbors. It cannot require *a priori* density information, as density will change due to environmental effects and node failure.

In this paper, we propose Trickle, an algorithm for code propagation and maintenance in wireless sensor networks. Borrowing techniques from the epidemic, scalable multicast, and wireless broadcast literatures, Trickle regulates itself using a local "polite gossip" to exchange code metadata (we defer a detailed discussion of Trickle with regards to this prior work to Section 6). Each mote periodically broadcasts metadata describing what code it has. However, if a mote hears gossip about identical metadata to its own, it stays quiet. When a mote hears old gossip, it triggers a code update, so the gossiper can be brought up to date. To achieve both rapid propagation and a low maintenance overhead, motes adjust the length of their gossiping attention spans, communicating more often when there is new code.

Trickle meets the three requirements. It imposes a maintenance overhead on the order of a few packets an hour (which can easily be pushed lower), propagates updates across multi-hop networks in tens of seconds, and scales to thousand-fold changes in network density. In addition, it handles network repopulation, is robust to network transience, loss, and disconnection, and requires very little state (in our implementation, eleven bytes).

In Section 2, we outline the experimental methodologies of this study. In Section 3, we describe the basic primitive of Trickle and its conceptual basis. In Section 4, we present Trickle's maintenance algorithm, evaluating its scalability with regards to network density. In Section 5, we show how the maintenance algorithm can be modified slightly to enable rapid propagation, and evaluate how quickly Trickle propagates code. We review related work in Section 6, and conclude in Section 7.

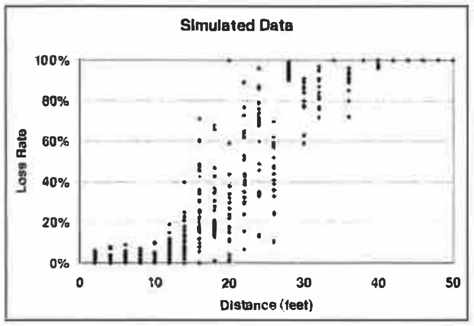


Figure 1: TOSSIM Packet Loss Rates over Distance

2. METHODOLOGY

We use three different platforms to investigate and evaluate Trickle. The first is a high-level, abstract algorithmic simulator written especially for this study. The second is TOSSIM [14], a bit-level mote simulator for TinyOS, a sensor network operating system [11]. TOSSIM compiles directly from TinyOS code. Finally, we used TinyOS mica-2 motes for empirical studies, to validate our simulation results and prove the real-world effectiveness of Trickle. The same implementation of Trickle ran on motes and in TOSSIM.

2.1 Abstract Simulation

To quickly evaluate Trickle under controlled conditions, we implemented a Trickle-specific algorithmic simulator. Little more than an event queue, it allows configuration of all of Trickle's parameters, run duration, the boot time of motes, and a uniform packet loss rate (same for all links) across a single hop network. Its output is a packet send count.

2.2 TOSSIM

The TOSSIM simulator compiles directly from TinyOS code, simulating complete programs from application level logic to the network at a bit level [14]. It simulates the implementation of the entire TinyOS network stack, including its CSMA protocol, data encodings, CRC checks, collisions, and packet timing. TOSSIM models mote connectivity as a directed graph, where vertices are motes and edges are links; each link has a bit error rate, and as the graph is directed, link error rates can be asymmetric. This occurs when only one direction has good connectivity, a phenomenon that several empirical studies have observed [7, 23, 3]. The networking stack (based on the mica platform implementation) can handle approximately forty packets per second, with each carrying a 36 byte payload.

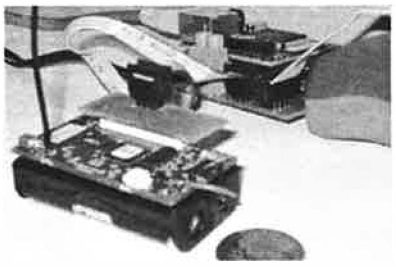


Figure 2: The TinyOS mica2

To generate network topologies, we used TOSSIM's empirical model, based on data gathered from TinyOS motes [7]. Figure 1 shows an experiment illustrating the model's packet loss rates over distance (in feet). As link directions are sampled independently, intermediate distances such as twenty feet commonly exhibit link asymmetry. Physical topologies are fed into the loss distribution function, producing a loss topology. In our studies, link error rates were constant for the duration of a simulation, but packet loss rates could be affected by dynamic interactions such as collisions at a receiver.

In addition to standard bit-level simulations, we used a modified version of TOSSIM that supports packet-level simulations. This version simulates loss due to packet corruption from bit errors, but does not model collisions. By comparing the results of the full bit-level simulation and this simpler packet-level simulation, we can ascertain when packet collisions – failures of the underlying MAC – are the cause of protocol behavior. In this paper, we refer to the full TOSSIM simulation as TOSSIM-bit, and the packet level simulation as TOSSIM-packet.

2.3 TinyOS motes

In our empirical experiments, we used TinyOS mica2 motes, with a 916MHz radio.[1] These motes provide 128KB of program memory, 4KB of RAM, and a 7MHz 8-bit microcontroller for a processor. The radio transmits at 19.2 Kbit, which after encoding and media access, is approximately forty TinyOS packets/second, each with a thirty-six byte data payload. For propagation experiments, we instrumented mica2 motes with a special hardware device that bridges their UART to TCP; other computers can connect to the mote with a TCP socket to read and write data to the mote. We used this to obtain millisecond granularity timestamps on network events. Figure 2 shows a picture of one of the mica2 motes used in our experiments.

We performed two empirical studies. One involved placing varying number of motes on a table, with the transmission strength set very low to create a small multihop network. The other was a nineteen mote network in an office area, approximately 160' by 40'. Section 5 presents the latter experiment in greater depth.

3. TRICKLE OVERVIEW

In the next three sections, we introduce and evaluate Trickle. In this section, we describe the basic algorithm primitive colloquially, as well as its conceptual basis. In Section 4, we describe the algorithm more formally, and evaluate the scalability of Trickle's maintenance cost, starting with an ideal case – a lossless and perfectly synchronized single-hop network. Incrementally, we remove each of these three constraints, quantifying scalability in simulation and validating the simulation results with an empirical study. In Section 5, we show how, by adjusting the length of time intervals, Trickle's maintenance algorithm can be easily adapted to also rapidly propagate code while imposing a minimal overhead. Trickle assumes that motes can succinctly describe their code with metadata, and by comparing two different pieces of metadata can determine which mote needs an update.

Trickle's basic primitive is simple: every so often, a mote transmits code metadata if it has not heard a few other motes transmit the same thing. This allows Trickle to scale to thousand-fold variations in network density, quickly propagate updates, distribute transmission load evenly, be robust to transient disconnections, handle network repopulations, and impose a maintenance overhead on the order of a few packets per hour per mote.

Trickle sends all messages to the local broadcast address. There are two possible results to a Trickle broadcast: either every mote that hears the message is up to date, or a recipient detects the need for an update. Detection can be the result of either an out-of-date mote hearing someone has new code, or an updated mote hearing someone has old code. As long as every mote communicates somehow – either receives or transmits – the need for an update will be detected.

For example, if mote A broadcasts that it has code ϕ_x, but B has code ϕ_{x+1}, then B knows that A needs an update. Similarly, if B broadcasts that it has ϕ_{x+1}, A knows that it needs an update. If B broadcasts updates, then all of its neighbors can receive them without having to advertise their need. Some of these recipients might not even have heard A's transmission.

In this example, it does not matter who first transmits, A or B; either case will detect the inconsistency. All that matters is that some motes communicate with one another at some nonzero rate; we will informally call this the "communication rate." As long as the network is connected and there is some minimum communication rate for each mote, everyone will stay up to date.

The fact that communication can be either transmission or reception enables Trickle to operate in sparse as well as dense networks. A single, disconnected mote

[1]There is also a 433 MHz variety.

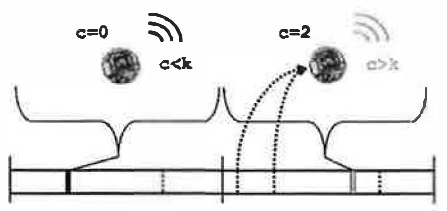


Figure 3: Trickle Maintenance with a k of 1. *Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions. Solid lines mark interval boundaries. Both I_1 and I_2 are of length τ.*

must transmit at the communication rate. In a lossless, single-hop network of size n, the sum of transmissions over the network is the communication rate, so for each mote it is $\frac{1}{n}$. Sparser networks require more transmissions per mote, but utilization of the radio channel *over space* will not increase. This is an important property in wireless networks, where the channel is a valuable shared resource. Additionally, reducing transmissions in dense networks conserves system energy.

We begin in Section 4 by describing Trickle's maintenance algorithm, which tries to keep a constant communication rate. We analyze its performance (in terms of transmissions and communication) in the idealized case of a single-hop lossless network with perfect time synchronization. We relax each of these assumptions by introducing loss, removing synchronization, and using a multi-hop network. We show how each relaxation changes the behavior of Trickle, and, in the case of synchronization, modify the algorithm slightly to accommodate.

4. MAINTENANCE

Trickle uses "polite gossip" to exchange code metadata with nearby network neighbors. It breaks time into intervals, and at a random point in each interval, it considers broadcasting its code metadata. If Trickle has already heard several other motes gossip the same metadata in this interval, it politely stays quiet: repeating what someone else has said is rude.

When a mote hears that a neighbor is behind the times (it hears older metadata), it brings everyone nearby up to date by broadcasting the needed pieces of code. When a mote hears that it is behind the times, it repeats the latest news it knows of (its own metadata); following the first rule, this triggers motes with newer code to broadcast it.

More formally, each mote maintains a counter c, a threshold k, and a timer t in the range $[0, \tau]$. k is a small, fixed integer (e.g., 1 or 2) and τ is a time constant. We discuss the selection of τ in depth in Section 5. When a mote hears metadata identical to its own, it increments c. At time t, the mote broadcasts its metadata if $c < k$. When the interval of size τ completes, c is reset to zero and t is reset to a new random value in the range $[0, \tau]$. If a mote with code ϕ_x hears a summary for ϕ_{x-y}, it broadcasts the code necessary to bring ϕ_{x-y} up to ϕ_x. If it hears a summary for ϕ_{x+y}, it broadcasts its own summary, triggering the mote with ϕ_{x+y} to send updates.

Figure 3 has a visualization of Trickle in operation on a single mote for two intervals of length τ with a k of 1 and no new code. In the first interval, I_1, the mote does not hear any transmissions before its t, and broadcasts. In the second interval, I_2, it hears two broadcasts of metadata identical to its, and so suppresses its broadcast.

Using the Trickle algorithm, each mote broadcasts a summary of its data at most once per period τ. If a mote hears k motes with the same program before it transmits, it suppresses its own transmission. In perfect network conditions – a lossless, single-hop topology – there will be k transmissions every τ. If there are n motes and m non-interfering single-hop networks, there will be km transmissions, which is independent of n. Instead of fixing the per-mote send rate, Trickle dynamically regulates its send rate to the network density to meet a communication rate, requiring no a priori assumptions on the topology. In each interval τ, the sum of receptions and sends of each mote is k.

The random selection of t uniformly distributes the choice of who broadcasts in a given interval. This evenly spreads the transmission energy load across the network. If a mote with n neighbors needs an update, the expected latency to discover this from the beginning of the interval is $\frac{\tau}{n+1}$. Detection happens either because the mote transmits its summary, which will cause others to send updates, or because another mote transmits a newer summary. A large τ has a lower energy overhead (in terms of packet send rate), but also has a higher discovery latency. Conversely, a small τ sends more messages but discovers updates more quickly.

This km transmission count depends on three assumptions: no packet loss, perfect interval synchronization, and a single-hop network. We visit and then relax each of these assumptions in turn. Discussing each assumption separately allows us to examine the effect of each, and in the case of interval synchronization, helps us make a slight modification to restore scalability.

4.1 Maintenance with Loss

The above results assume that motes hear every transmission; in real-world sensor networks, this is rarely the case. Figure 4 shows how packet loss rates affect the number of Trickle transmissions per interval in a single-hop network as density increases. These results are from the abstract simulator, with $k = 1$. Each line is a uniform loss rate for all node pairs. For a given rate, the number of transmissions grows with density at $O(log(n))$.

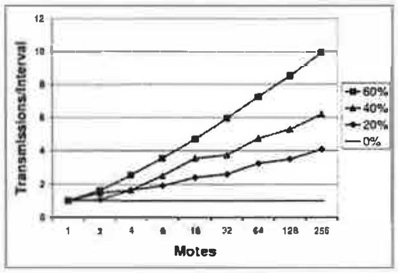


Figure 4: Number of Transmissions as Density Increases for Different Packet Loss Rates.

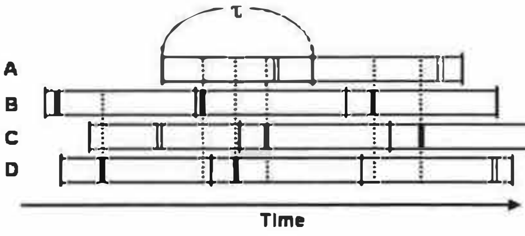


Figure 5: The Short Listen Problem For Motes A, B, C, and D. *Dark bars represent transmissions, light bars suppressed transmissions, and dashed lines are receptions. Tick marks indicate interval boundaries. Mote B transmits in all three intervals.*

This logarithmic behavior represents the probability that a single mote misses a number of transmissions. For example, with a 10% loss rate, there is a 10% chance a mote will miss a single packet. If a mote misses a packet, it will transmit, resulting in two transmissions. There is correspondingly a 1% chance it will miss two, leading to three transmissions, and a 0.1% chance it will miss three, leading to four. In the extreme case of a 100% loss rate, each mote is by itself: transmissions scale linearly.

Unfortunately, to maintain a per-interval minimum communication rate, this logarithmic scaling is inescapable: $O(log(n))$ is the best-case behavior. The increase in communication represents satisfying the requirements of the worst case mote; in order to do so, the expected case must transmit a little bit more. Some motes don't hear the gossip the first time someone says it, and need it repeated. In the rest of this work, we consider $O(log(n))$ to be the desired scalability.

4.2 Maintenance without Synchronization

The above results assume that all motes have synchronized intervals. Inevitably, time synchronization imposes a communication, and therefore energy, overhead. While some networks can provide time synchronization to Trickle, others cannot. Therefore, Trickle should be able to work in the absence of this primitive.

Unfortunately, without synchronization, Trickle can suffer from the *short-listen* problem. Some subset of motes

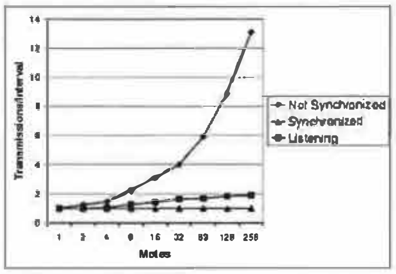


Figure 6: The Short Listen Problem's Effect on Scalability, $k = 1$. *Without synchronization, Trickle scales with $O(\sqrt{n})$. A listening period restores this to asymptotically bounded by a constant.*

gossip soon after the beginning of their interval, listening for only a short time, before anyone else has a chance to speak up. If all of the intervals are synchronized, the first gossip will quiet everyone else. However, if not synchronized, it might be that a mote's interval begins just after the broadcast, and it too has chosen a short listening period. This results in redundant transmissions.

Figure 5 shows an instance of this phenomenon. In this example, mote B selects a small t on each of its three intervals. Although other motes transmit, mote B never hears those transmissions before its own, and its transmissions are never suppressed. Figure 6 shows how the short-listen problem effects the transmission rate in a lossless network with $k = 1$. A perfectly synchronized single-hop network scales perfectly, with a constant number of transmissions. In a network without any synchronization between intervals, however, the number of transmissions per interval increases significantly.

The short-listen problem causes the number of transmissions to scale as $O(\sqrt{n})$ with network density. [2] Unlike loss, where extra $O(log(n))$ transmissions are sent to keep the worst case mote up to date, the additional transmissions due to a lack of synchronization are completely redundant, and represent avoidable inefficiency.

To remove the short-listen effect, we modified Trickle slightly. Instead of picking a t in the range $[0, \tau]$, t is selected in the range $[\frac{\tau}{2}, \tau]$, defining a "listen-only" period of the first half of an interval. Figure 7 depicts the modified algorithm. A listening period improves scalability by enforcing a simple constraint. If sending a message guarantees a silent period of some time T that is inde-

[2]To see this, assume the network of n motes with an interval τ is in a steady state. If interval skew is uniformly distributed, then the expectation is that one mote will start its interval every $\frac{\tau}{n}$. For time t after a transmission, $\frac{nt}{\tau}$ will have started their intervals. From this, we can compute the expected time after a transmission that another transmission will occur. This is when

$$\prod_{t=0}^{n} (1 - \frac{t}{n}) < \frac{1}{2}$$

which is when $t \approx \sqrt{n}$, that is, when $\frac{\sqrt{n}}{\tau}$ time has passed. There will therefore be $O(\sqrt{n})$ transmissions.

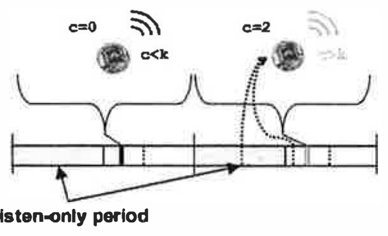


Figure 7: Trickle Maintenance with a k of 1 and a Listen-Only Period. *Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions.*

pendent of density, then the send rate is bounded above (independent of the density). When a mote transmits, it suppresses all other motes for at least the length of the listening period. With a listen period of $\frac{\tau}{2}$, it bounds the total sends in a lossless single-hop network to be $2k$, and with loss scales as $2k \cdot log(n)$, returning scalability to the $O(log(n))$ goal.

The "Listening" line in Figure 6 shows the number of transmissions in a single-hop network with no synchronization when Trickle uses this listening period. As the network density increases, the number of transmissions per interval asymptotically approaches two. The listening period does not harm performance when the network is synchronized: there are k transmissions, but they are all in the second half of the interval.

To work properly, Trickle needs a source of randomness; this can come from either the selection of t or from a lack of synchronization. By using both sources, Trickle works in either circumstance, or any point between the two (e.g., partial or loose synchronization).

4.3 Maintenance in a Multi-hop Network

To understand Trickle's behavior in a multi-hop network, we used TOSSIM, randomly placing motes in a 50'x50' area with a uniform distribution, a τ of one second, and a k of 1. To discern the effect of packet collisions, we used both TOSSIM-bit and TOSSIM-packet (the former models collisions, and the latter does not). Drawing from the loss distributions in Figure 1, a 50'x50' grid is a few hops wide. Figure 8 shows the results of this experiment.

Figure 8(a) shows how the number of transmissions per interval scales as the number of motes increases. In the absence of collisions, Trickle scales as expected, at $O(log(n))$. This is also true in the more accurate TOSSIM-bit simulations for low to medium densities; however, once there is over 128 motes, the number of transmissions increases significantly.

This result is troubling – it suggests that Trickle cannot scale to very dense networks. However, this turns out to be a limitation of TinyOS's CSMA as network utilization increases, and not Trickle itself. Figure 8(b) shows the average number of receptions per transmission for the same experiments. Without packet collisions, as network density increases exponentially, so does the reception/transmission ratio. Packet collisions increase loss, and therefore the base of the logarithm in Trickle's $O(log(n$ scalability. The increase is so great that Trickle's aggregate transmission count begins to scale linearly. As the number of transmissions over space increases, so does the probability that two will collide.

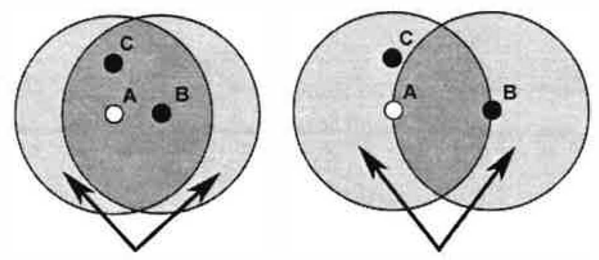
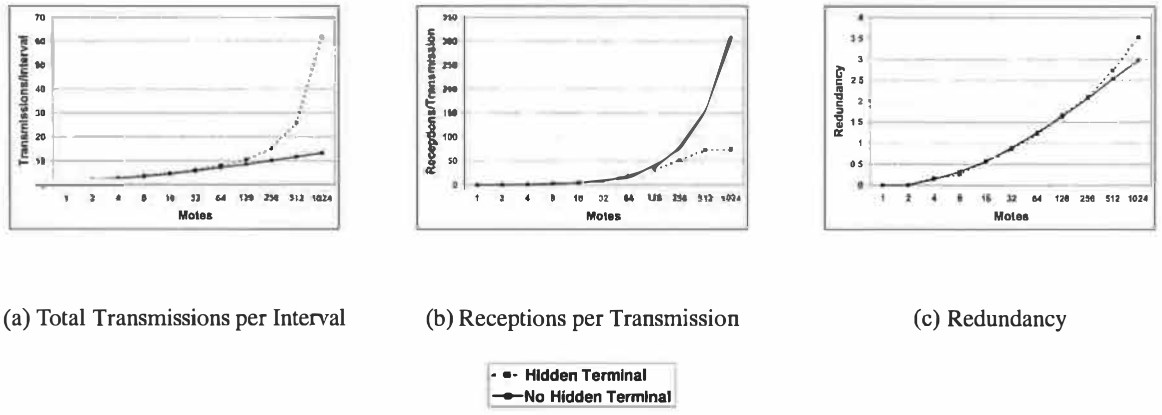


Figure 9: **The Effect of Proximity on the Hidden Terminal Problem.** *When C is within range of both A and B, CSMA will prevent C from interfering with transmissions between A and B. But when C is in range of A but not B, B might start transmitting without knowing that C is already transmitting, corrupting B's transmission. Note that when A and B are farther apart, the region where C might cause this "hidden terminal" problem is larger.*

As the network becomes very dense, it succumbs to the *hidden terminal problem*, a known issue with CSMA protocols. In the classic hidden terminal situation, there are three nodes, a, b, and c, with effective carrier sense between a and b and a and c. However, as b and c do not hear one another, a CSMA protocol will let them transmit at the same time, colliding at b, who will hear neither. In this situation, c is a hidden terminal to b and vice versa. Figure 9 shows an instance of this phenomenon in a simplistic disk model.

In TOSSIM-bit, the reception/transmission ratio plateaus around seventy-five: each mote thinks it has about seventy-five one-hop network neighbors. At high densities, many packets are being lost due to collisions due to the hidden terminal problem. In the perfect scaling model, the number of transmissions for m isolated and independent single-hop networks is mk. In a network, there is a *physical* density (defined by the radio range), but the hidden terminal problem causes motes to lose packets; hearing less traffic, they are aware of a smaller *observed* density. Physical density represents the number of motes who can hear a transmission in the absence of any other traffic, while observed density is a function of other, possibly conflicting, traffic in the network. Increasing physical density also make collision more likely; observed density does not necessarily increase at the same rate.



(a) Total Transmissions per Interval (b) Receptions per Transmission (c) Redundancy

Figure 8: Simulated Trickle Scalability for a Multi-hop Network with Increasing Density. *Motes were uniformly distributed in a 50'x50' square area.*

When collisions make observed density lower than physical density, the set of motes observed to be neighbors is tied to physical proximity. The set of motes that can interfere with communication by the hidden terminal problem is larger when two motes are far away than when they are close. Figure 9 depicts this relationship.

Returning to Figure 8(b), from each mote's perspective in the 512 and 1024 mote experiments, the observed density is seventy-five neighbors. This does not change significantly as physical density increases. As a mote that can hear n neighbors, ignoring loss and other complexities, will broadcast in an interval with probability $\frac{1}{n}$, the lack of increase in observed density increases the number of transmissions (e.g., $\frac{512}{75} \rightarrow \frac{1024}{75}$).

TOSSIM simulates the mica network stack, which can handle approximately forty packets a second. As utilization reaches a reasonable fraction of this (e.g., 10 packets/second, with 128 nodes), the probability of a collision becomes significant enough to affect Trickle's behavior. As long as Trickle's network utilization is low, it scales as expected. However, increased utilization affects connectivity patterns, so that Trickle must transmit more than in an quiet network. The circumstances of Figure 8, very dense networks and a tiny interval, represent a corner case. As we present in Section 5, maintenance intervals are more likely to be on the order of tens of minutes. At these interval sizes, network utilization will never grow large as long as k is small.

To better understand Trickle in multi-hop networks, we use the metric of *redundancy*. Redundancy is the portion of messages heard in an interval that were unnecessary communication. Specifically, it is each mote's expected value of $\frac{c+s}{k} - 1$, where s is 1 if the mote transmitted and 0 if not. A redundancy of 0 means Trickle works perfectly; every mote communicates k times. For example, a mote with a k of 2, that transmitted ($s = 1$), and then received twice ($c = 2$), would have a redundancy of 0.5 ($\frac{2+1}{2} - 1$): it communicated 50% more than the optimum of k.

Redundancy can be computed for the single-hop experiments with uniform loss (Figures 4 and 6). For example, in a single-hop network with a uniform 20% loss rate and a k of 1, 3 transmissions/interval has a redundancy of 1.4= $((3 \cdot 0.8) - 1)$, as the expectation is that each mote receives 2.4 packets, and three motes transmit.

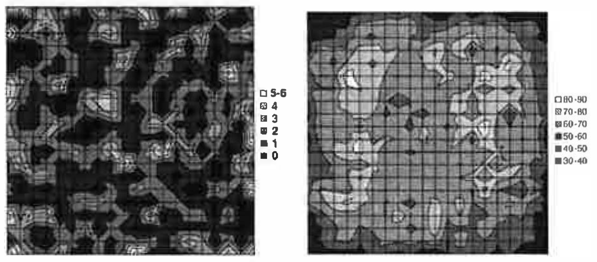
Figure 8(c) shows a plot of Trickle redundancy as network density increases. For a one-thousand mote– larger than any yet deployed – multi-hop network, in the presence of link asymmetry, variable packet loss, and the hidden terminal problem, the redundancy is just over 3.

Redundancy grows with a simple logarithm of the observed density, and is due to the simple problem outlined in Section 4.1: packets are lost. To maintain a communication rate for the worst case mote, the average case must communicate a little bit more. Although the communication increases, the actual per-mote transmission rate shrinks. Barring MAC failures, Trickle scales as hoped – $O(log(n))$ – in multi-hop networks.

4.4 Load Distribution

One of the goals of Trickle is to impose a low overhead. The above simulation results show that few packets are sent in a network. However, this raises the question of which motes sent those packets; 500 transmissions evenly distributed over 500 motes does not impose a high cost, but 500 messages by one mote does.

Figure 10(a) shows the transmission distribution for a simulated 400 mote network in a 20 mote by 20 mote grid with a 5 foot spacing (the entire grid was 95'x95'), run in TOSSIM-bit. Drawing from the empirical distri-



(a) Transmissions (b) Receptions

Figure 10: Communication topography of a simulated 400 mote network in a 20x20 grid with 5 foot spacing (95'x95'), running for twenty minutes with a τ of one minute. *The x and y axes represent space, with motes being at line intersections. Color denotes the number of transmissions or receptions at a given mote.*

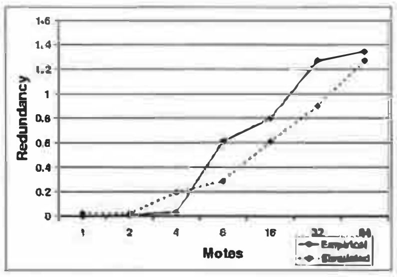


Figure 11: Empirical and Simulated over Density. *The simulated data is the same as Figure 8.*

Event	Action
τ Expires	Double τ, up to τ_h. Reset c, pick a new t.
t Expires	If $c < k$, transmit.
Receive same metadata	Increment c.
Receive newer metadata	Set τ to τ_l. Reset c, pick a new t.
Receive newer code	Set τ to τ_l. Reset c, pick a new t.
Receive older metadata	Send updates.

t is picked from the range $[\frac{\tau}{2}, \tau]$

Figure 12: Trickle Pseudocode.

butions in Figure 1, a five foot spacing forms a six hop network from grid corner to corner. This simulation was run with a τ of one minute, and ran for twenty minutes of virtual time. The topology shows that some motes send more than others, in a mostly random pattern. Given that the predominant range is one, two, or three packets, this non-uniformity is easily attributed to statistical variation. A few motes show markedly more transmissions, for example, six. This is the result of some motes being poor receivers. If many of their incoming links have high loss rates (drawn from the distribution in Figure 1), they will have a small observed density, as they receive few packets.

Figure 10(b) shows the reception distribution. Unlike the transmission distribution, this shows clear patterns. motes toward the edges and corners of the grid receive fewer packets than those in the center. This is due to the non-uniform network density; a mote at a corner has one quarter the neighbors as one in the center. Additionally, a mote in the center has many more neighbors that cannot hear one another; so that a transmission in one will not suppress a transmission in another. In contrast, almost all of the neighbors of a corner mote can hear one another. Although the transmission topology is quite noisy, the reception topography is smooth. The number of transmissions is very small compared to the number of receptions: the communication rate across the network is fairly uniform.

4.5 Empirical Study

To evaluate Trickle's scalability in a real network, we recreated, as best we could, the experiments shown in Figures 6 and 8. We placed motes on a small table, with their transmission signal strength set very low, making the table a small multi-hop network. With a τ of one minute, we measured Trickle redundancy over a twenty minute period for increasing numbers of motes. Figure 11 shows the results. They show similar scaling to the results from TOSSIM-bit. For example, the TOSSIM-bit results in Figure 8(c) show a 64 mote network having an redundancy of 1.1; the empirical results show 1.35. The empirical results show that maintenance scales as the simulation results indicate it should: logarithmically.

The above results quantified the maintenance overhead. Evaluating propagation requires an implementation; among other things, there must be code to propagate. In the next section, we present an implementation of Trickle, evaluating it in simulation and empirically.

5. PROPAGATION

A large τ (gossiping interval) has a low communication overhead, but slowly propagates information. Conversely, a small τ has a higher communication overhead, but propagates more quickly. These two goals, rapid propagation and low overhead, are fundamentally at odds: the former requires communication to be frequent, while the latter requires it to be infrequent.

By dynamically scaling τ, Trickle can use its maintenance algorithm to rapidly propagate updates with a very small cost. τ has a lower bound, τ_l, and an upper bound τ_h. When τ expires, it doubles, up to τ_h. When a mote hears a summary with newer data than it has, it resets τ to be τ_l. When a mote hears a summary with older code than it has, it sends the code, to bring the other mote up to date. When a mote installs new code, it resets τ to τ_l, to make sure that it spreads quickly. This is necessary

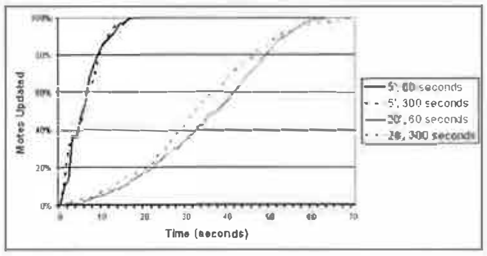
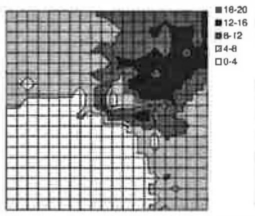
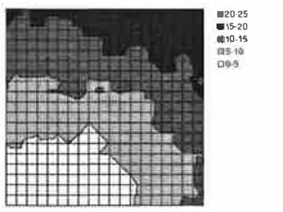


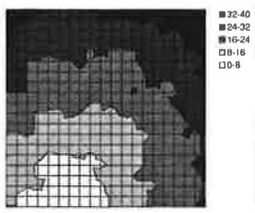
Figure 13: Simulated Code Propagation Rate for Different τ_hs.



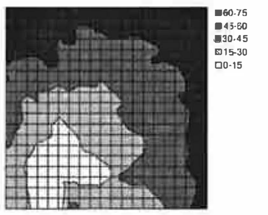
(a) 5' Spacing, 6 hops



(b) 10' Spacing, 16 hops



(c) 15' Spacing, 32 hops



(d) 20' Spacing, 40 hops

Figure 14: Simulated Time to Code Propagation Topography in Seconds. *The hop count values in each legend are the expected number of transmissions necessary to get from corner to corner, considering loss.*

for when a mote receives code it did not request, that is, didn't reset its τ for. Figure 12 shows pseudocode for this complete version of Trickle.

Essentially, when there's nothing new to say, motes gossip infrequently: τ is set to τ_h. However, as soon as a mote hears something new, it gossips more frequently, so those who haven't heard it yet find out. The chatter then dies down, as τ grows from τ_l to τ_h.

By adjusting τ in this way, Trickle can get the best of both worlds: rapid propagation, and low maintenance overhead. The cost of a propagation event, in terms of additional sends caused by shrinking τ, is approximately $log(\frac{\tau_h}{\tau_l})$. For a τ_l of one second and a τ_h of one hour, this is a cost of eleven packets to obtain a three-thousand fold increase in propagation rate (or, from the other perspective, a three thousand fold decrease in maintenance overhead). The simple Trickle policy, "every once in a while, transmit unless you've heard a few other transmissions," can be used both to inexpensively maintain code and quickly propagate it.

We evaluate an implementation of Trickle, incorporated into Maté, a tiny bytecode interpreter for TinyOS sensor networks [13]. We first present a brief overview of Maté and its Trickle implementation. Using TOSSIM, we evaluate how how rapidly Trickle can propagate an update through reasonably sized (i.e., 400 mote) networks of varying density. We then evaluate Trickle's propagation rate in a small (20 mote) real-world network.

5.1 Maté, a Trickle Implementation

Maté has a small, static set of code routines. Each routine can have many versions, but the runtime only keeps the most recent one. By replacing these routines, a user can update a network's program. Each routine fits in a single TinyOS packet and has a version number. The runtime installs routines with a newer version number when it receives them.

Instead of sending entire routines, motes can broadcast version summaries. A version summary contains the version numbers of all of the routines currently installed. A mote determines that someone else needs an update by hearing that they have an older version.

Maté uses Trickle to periodically broadcast version summaries. In all experiments, code routines fit in a single TinyOS packet (30 bytes). The runtime registers routines with a propagation service, which then maintains all of the necessary timers and broadcasts, notifying the runtime when it installs new code. The actual code propagation mechanism is outside the scope of Trickle, but we describe it here for completeness. When a mote hears an older vector, it broadcasts the missing routines three times: one second, three seconds, and seven seconds after hearing the vector. If code transmission redundancy were a performance issue, it could also use Trickle's suppression mechanism. For the purpose of our experiments, however, it was not.

The Maté implementation maintains a 10Hz timer, which it uses to increment a counter. t and τ are represented in ticks of this 10Hz clock. Given that the current mote platforms can transmit on the order of 40 packets/second, we found this granularity of time to be sufficient. If the power consumption of maintaining a 10Hz clock were an issue (as it may be in some deployments), a non-periodic implementation could be used instead.

5.2 Simulation

We used TOSSIM-bit to observe the behavior of Trickle during a propagation event. We ran a series of simula-

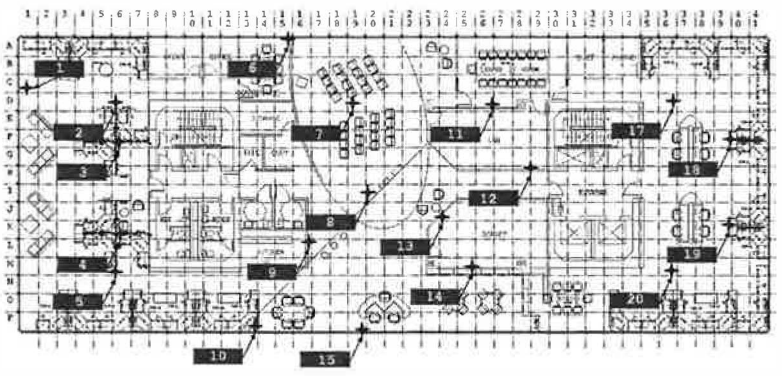
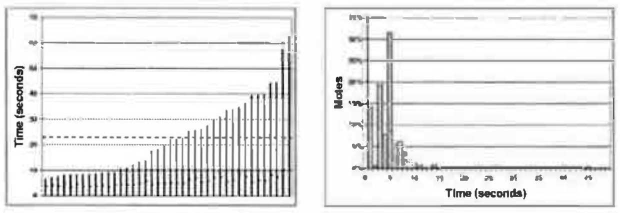
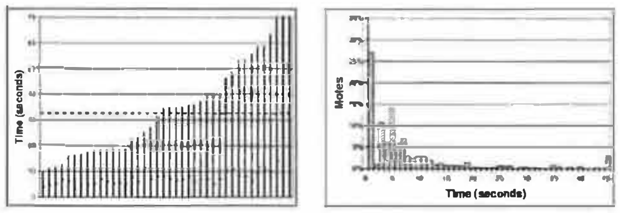


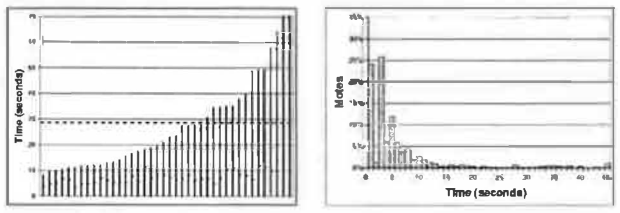
Figure 15: Empirical Testbed



(a) τ_h of 1 minute, $k = 1$



(b) τ_h of 20 minutes, $k = 1$



(c) τ_h of 20 minutes, $k = 2$

Figure 16: Empirical Network Propagation Time. *The graphs on the left show the time to complete reprogramming for 40 experiments, sorted with increasing time. The graphs on the right show the distribution of individual mote reprogramming times for all of the experiments.*

tions, each of which had 400 motes regularly placed in a 20x20 grid, and varied the spacing between motes. By varying network density, we could examine how Trickle's propagation rate scales over different loss rates and physical densities. Density ranged from a five foot spacing between motes up to twenty feet (the networks were 95'x95' to 380'x380'). We set τ_l to one second and τ_h to one minute. From corner to corner, these topologies range from six to forty hops. [3]

The simulations ran for five virtual minutes. motes booted with randomized times in the first minute, selected from a uniform distribution. After two minutes, a mote near one corner of the grid advertised a new Maté routine. We measured the propagation time (time for the last mote to install the new routine from the time it first appeared) as well as the topographical distribution of routine installation time. The results are shown in Figures 13 and 14. Time to complete propagation varied from 16 seconds in the densest network to about 70 seconds for the sparsest. Figure 13 shows curves for only the 5' and 20' grids; the 10' and 15' grid had similar curves.

Figure 14(a) shows a manifestation of the hidden terminal problem. This topography doesn't have the wave pattern we see in the experiments with sparser networks. Because the network was only a few hops in area, motes near the edges of the grid were able to receive and install the new capsule quickly, causing their subsequent transmissions to collide in the upper right corner. In contrast, the sparser networks exhibited a wave-like propagation because the sends mostly came from a single direction throughout the propagation event.

Figure 13 shows how adjusting τ_h changes the propagation time for the five and twenty foot spacings. Increasing τ_h from one minute to five does not significantly

[3]These hop count values come from computing the minimum cost path across the network loss topology, where each link has a weight of $\frac{1}{1-loss}$, or the expected number of transmissions to successfully traverse that link.

affect the propagation time; indeed, in the sparse case, it propagates faster until roughly the 95th percentile. This result indicates that there may be little trade-off between the maintenance overhead of Trickle and its effectiveness in the face of a propagation event.

A very large τ_h can increase the time to discover inconsistencies to be approximately $\frac{\tau_h}{2}$. However, this is only true when two stable subnets ($\tau = \tau_h$) with different code reconnect. If new code is introduced, it immediately triggers motes to τ_l, bringing the network to action.

5.3 Empirical Study

As Trickle was implemented as part of Maté, several other services run concurrently with it. The only one of possible importance is the ad-hoc routing protocol, which periodically sends out network beacons to estimate link qualities. However, as both Trickle packets and these beacons are very infrequent compared to channel capacity (e.g., at most 1 packet/second), this does not represent a significant source of noise.

We deployed a nineteen mote network in an office area, approximately 160' by 40'. We instrumented fourteen of the motes with the TCP interface described in Section 2, for precise timestamping. When Maté installed a new piece of code, it sent out a UART packet; by opening sockets to all of the motes and timestamping when this packet is received, we can measure the propagation of code over a distributed area.

Figure 15 shows a picture of the office space and the placement of the motes. motes 4, 11, 17, 18 and 19 were not instrumented; motes 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, and 20 were. mote 16 did not exist.

As with the above experiments, Trickle was configured with a τ_l of one second and a τ_h of one minute. The experiments began with the injection of a new piece of code through a TinyOS GenericBase, which is a simple bridge between a PC and a TinyOS network. The GenericBase broadcast the new piece of code three times in quick succession. We then logged when each mote had received the code update, and calculated the time between the first transmission and installation.

The left hand column of Figure 16 shows the results of these experiments. Each bar is a separate experiment (40 in all). The worst-case reprogramming time for the instrumentation points was just over a minute; the best case was about seven seconds. The average, shown by the dark dotted line, was just over twenty-two seconds for a τ_h of sixty seconds (Figure 16(a)), while it was thirty-two seconds for a τ_h of twenty minutes (Figure 16(b)).

The right hand column of Figure 16 shows a distribution of the time to reprogramming for individual motes across all the experiments. This shows that almost all motes are reprogrammed in the first ten seconds: the longer times in Figure 16 are from the very long tail on this distribution. The high loss characteristics of the mote radio, combined with t's exponential scaling, make this an issue. When scaling involves sending only a handful (e.g., $log_2(60)$) of packets in a neighborhood in order to conserve energy, long tails are inevitable.

In Figure 16, very few motes reprogram between one and two seconds after code is introduced. This is an artifact of the granularity of the timers used, the capsule propagation timing, and the listening period. Essentially, from the first broadcast, three timers expire: $[\frac{\tau_l}{2}, \tau_l]$ for motes with the new code,$[\frac{\tau_l}{2}, \tau_l]$ for motes saying they have old code, then one second before the first capsule is sent. This is approximately $2 \cdot \frac{\tau_l}{2} + 1$; with a τ_l of one second, this latency is two seconds.

5.4 State

The Maté implementation of Trickle requires few system resources. It requires approximately seventy bytes of RAM; half of this is a message buffer for transmissions, a quarter is pointers to the Maté routines. Trickle itself requires only eleven bytes for its counters; the remaining RAM is used by coordinating state such as pending and initialization flags. The executable code is 2.5 KB; TinyOS's inlining and optimizations can reduce this by roughly 30%, to 1.8K. The algorithm requires few CPU cycles, and can operate at a very low duty cycle.

6. RELATED WORK

Trickle draws on two major areas of prior research. Both assume network characteristics distinct from low-power wireless sensor networks, such as cheap communication, end-to-end transport, and limited (but existing) loss. The first area is controlled, density-aware flooding algorithms for wireless and multicast networks [6, 16, 19]. The second is epidemic and gossiping algorithms for maintaining data consistency in distributed systems [2, 4, 5].

Prior work in network broadcasts has dealt with a different problem than the one Trickle tackles: delivering a piece of data to as many nodes as possible within a certain time period. Early work showed that in wireless networks, simple broadcast retransmission could easily lead to the broadcast storm problem [19], where competing broadcasts saturate the network. This observation led to work in probabilistic broadcasts [16, 21], and adaptive dissemination [9]. Just as with earlier work in bimodal epidemic algorithms [1], all of these algorithms approach the problem of making a best-effort attempt to send a message to all nodes in a network, then eventually stop.

For example, Ni et al. propose a counter-based algorithm to prevent the broadcast storm problem by suppressing retransmissions [19]. This algorithm operates on a single interval, instead of continuously. As results in Figure 16 show, the loss rates in the class of wireless sensor network we study preclude a single interval from being sufficient. Additionally, their studies were on lossless, disk-based network topologies; it is unclear how they would perform in the sort of connectivity observed in the real world [12].

This is insufficient for sensor network code propagation. For example, it is unclear what happens if a mote rejoins three days after the broadcast. For configurations or code, the new mote should be brought up to date. Using prior wireless broadcast techniques, the only way to do so is periodically rebroadcast to the entire network. This imposes a significant cost on the entire network. In contrast, Trickle locally distributes data where needed.

The problem of propagating data updates through a distributed system has similar goals to Trickle, but prior work has been based on traditional wired network models. Demers et al. proposed the idea of using epidemic algorithms for managing replicated databases [5], while the PlanetP project [4] uses epidemic gossiping for a a distributed peer-to-peer index. Our techniques and mechanisms draw from these efforts. However, while traditional gossiping protocols use unicast links to a random member of a neighbor set, or based on a routing overlay [2], Trickle uses only a local wireless broadcast, and its mechanisms are predominantly designed to address the complexities that result.

Gossiping through the exchange of metadata is reminiscent of SPIN's three-way handshaking protocol [9]; the Impala system, deployed in ZebraNet, uses a similar approach [15]. Specifically, Trickle is similar to SPIN-RL, which works in broadcast environments and provides reliability in lossy networks. Trickle differs from and builds on SPIN in three major ways. First, the SPIN protocols are designed for transmitting when they detect an update is needed; Trickle's purpose is to perform that detection. Second, the SPIN work points out that periodically re-advertising data can improve reliability, but does not suggest a policy for doing so; Trickle is such a policy. Finally, the SPIN family, although connectionless, is session oriented. When a node A hears an advertisement from node B, it then requests the data from node B. In contrast, Trickle never considers addresses. Taking the previous example, with Trickle B sends an implicit request, which a node besides A may respond to.

Trickle's suppression mechanism is inspired by the request/repair algorithm used in Scalable and Reliable Multicast (SRM) [6]. However, SRM focuses on reliable delivery of data through a multicast group in a wired IP network. Using IP multicast as a primitive, SRM has a fully connected network where latency is a concern. Trickle adapts SRM's suppression mechanisms to the domain of multi-hop wireless sensor networks.

Although both techniques – broadcasts and epidemics – have assumptions that make them inappropriate to problem of code propagation and maintenance in sensor networks, they are powerful techniques that we draw from. An effective algorithm must adjust to local network density as controlled floods do, but continually maintain consistency in a manner similar to epidemic algorithms. Taking advantage of the broadcast nature of the medium, a sensor network can use SRM-like duplicate suppression to conserve precious transmission energy and scale to dense networks.

In the sensor network space, Reijers et al. propose energy efficient code distribution by only distributing changes to currently running code [20]. The work focuses on developing an efficient technique to compute and update changes to a code image through memory manipulation, but does not address the question of how to distribute the code updates in a network or how to validate that nodes have the right code. It is a program encoding that Trickle or a Trickle-like protocol can use to transmit updates.

The TinyDB sensor network query system uses an epidemic style of code forwarding [17]. However, it depends on periodic data collection with embedded metadata. Every tuple routed through the network has a query ID associated with it and a mote requests a new query when it hears it. In this case, the metadata has no cost, as it would be sent anyways. Also, this approach does not handle event-driven queries for rare events well; the query propagates when the event occurs, which may be too late.

7. DISCUSSION AND CONCLUSION

Using listen periods and dynamic τ values, Trickle meets the requirements set out in Section 1. It can quickly propagate new code into a network, while imposing a very small overhead. It does so using a very simple mechanism, and requires very little state. Scaling logarithmically with density, it can be used effectively in a wide range of networks. In one of our empirical experiments, Trickle imposes an overhead of less than three packets per hour, but reprograms the entire network in thirty seconds, with no effort from an end user.

A trade-off emerges between energy overhead and reprogramming rate. By using a dynamic communication rate, Trickle achieves a reprogramming rate comparable to frequent transmissions while keeping overhead comparable to infrequent transmissions. However, as Figure 16 shows, the exact relationship between constants such as τ_h and k is unclear in the context of these high loss networks. τ_l affects the head of the distribution , while τ_h affects the tail.

In this study, we have largely ignored the actual policy used to propagate code once Trickle detects the need to do so: Maté merely broadcasts code routines three times. Trickle suppression techniques can also be used to control the rate of code transmission. In the current Maté implementation, the blind code broadcast is a form of localized flood; Trickle acts as a flood control protocol. This behavior is almost the inverse of protocols such as SPIN [9], which transmits metadata freely but controls data transmission.

Assuming complete network propagation allows Trickle to decouple code advertisement from code transmission. As the protocol does not consider network addresses, the mote that advertises code – leading to an implicit request – may not be the one that transmits it. Instead of trying to enforce suppression on an abstraction of a logical group, which can become difficult in multi-hop networks, Trickle suppresses in terms of space, implicitly defining a group. Correspondingly, Trickle does not impose the overhead of discovering and maintaining logical groups, which can be significant.

One limitation of Trickle is that it currently assumes motes are always on. To conserve energy, long-term mote deployments often have very low duty cycles (e.g., 1%). Correspondingly, motes are rarely awake, and rarely able to receive messages. Communication scheduling schemes can define times for code communication, during which motes in the network wake up to run Trickle. Essentially, the Trickle time intervals become logical time, spread over all of the periods motes are actually awake. Understandably, this might require alternative tunings of τ_h and k. Trickle's scalability, however, stems from randomization and idle listening. As Section 4.3 showed, Trickle's transmission scalability suffers under a CSMA protocol as utilization increases. Another, and perhaps more promising, option is to use low power listening, where transmitters send very long start symbols so receivers can detect packets when sampling the channel at a very low rate [10]. For more dense networks, the receiver energy savings could make up for the transmitter energy cost.

Trickle was designed as a code propagation mechanism over an entire network, but it has greater applicability, and could be used to disseminate any sort of data. Additionally, one could change propagation scope by adding predicates to summaries, limiting the set of motes that consider them. For example, by adding a "hop count" predicate to local routing data, summaries of a mote's routing state could reach only two-hop network neighbors of the summary owner; this could be used to propagate copies of mote-specific information.

As sensor networks move from research to deployment, from laboratory to the real world, issues of management and reconfiguration will grow in importance. We have identified what we believe to be a core networking primitive in these systems, update distribution, and designed a scalable, lightweight algorithm to provide it.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599 and NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

8. REFERENCES

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[2] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 47–60. ACM Press, 2002.

[3] A. Cerpa, N. Busek, and D. Estrin. SCALE: A tool for simple connectivity assessment in lossy environments. Technical Report CENS-21, 2003.

[4] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. Technical Report DCS-TR-487, Department of Computer Science, Rutgers University, Sept. 2002.

[5] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[6] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 342–356. ACM Press, 1995.

[7] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks, 2002. Submitted for publication, February 2002.

[8] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Symposium on Operating Systems Principles*, pages 146–159, 2001.

[9] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185. ACM Press, 1999.

[10] J. Hill and D. E. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, nov/dec 2002.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[12] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dept. of Computer Science, Dartmouth College, July 2003.

[13] P. Levis and D. Culler. Maté: a Virtual Machine for Tiny Networked Sensors. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[14] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[15] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[16] J. Luo, P. Eugster, and J.-P. Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networks. In *Proc. of INFOCOM 2003*, 2003.

[17] S. R. Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, UC Berkeley, Decmeber 2003. `http://www.cs.berkeley.edu/~madden/thesis.pdf`.

[18] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[19] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[20] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*, 2003.

[21] Y. Sasson, D. Cavin, and A. Schiper. Probabilistic broadcast for flooding in wireless networks. Technical Report IC/2002/54, 2002.

[22] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN '04)*, January 2004.

[23] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First International Conference on Embedded Network Sensor Systems*, 2003.

Programming Sensor Networks Using Abstract Regions

Matt Welsh and Geoff Mainland
Harvard University
{mdw,mainland}@eecs.harvard.edu

Abstract

Wireless sensor networks are attracting increased interest for a wide range of applications, such as environmental monitoring and vehicle tracking. However, developing sensor network applications is notoriously difficult, due to extreme resource limitations of nodes, the unreliability of radio communication, and the necessity of low power operation. Our goal is to simplify application design by providing a set of programming primitives for sensor networks that abstract the details of low-level communication, data sharing, and collective operations.

We present *abstract regions*, a family of spatial operators that capture local communication within regions of the network, which may be defined in terms of radio connectivity, geographic location, or other properties of nodes. Regions provide interfaces for identifying neighboring nodes, sharing data among neighbors, and performing efficient reductions on shared variables. In addition, abstract regions expose the tradeoff between the accuracy and resource usage of communication operations. Applications can adapt to changing network conditions by tuning the energy and bandwidth usage of the underlying communication substrate. We present the implementation of abstract regions in the TinyOS programming environment, as well as results demonstrating their use for building adaptive sensor network applications.

1 Introduction

Sensor networks are an emerging computing platform consisting of large numbers of small, low-powered, wireless "motes" each with limited computation, sensing, and communication abilities. Sensor networks are being investigated for applications such as environmental monitoring [8, 27], seismic analysis of structures [7, 20], and tracking moving vehicles [29]. Still, sensor network programming is incredibly difficult, due to the limited capabilities and energy resources of each node as well as the unreliability of the radio channel.

As a result, application designers must make many complex, low-level choices, and build up a great deal of machinery to perform routing, time synchronization, node localization, and data aggregation. To date, little of this machinery has carried over directly from one application to the next, as it encapsulates application-specific tradeoffs in terms of complexity, resource usage, and communication patterns. In this paper, we investigate a suite of general-purpose communication primitives for sensor networks that provide addressing, data sharing, and reduction within local regions of the network. These primitives, which we call *abstract regions*, expose control over the resource consumption of communication, and provide feedback to applications on the accuracy and completeness of collective operations.

A key goal of sensor network programming is to save energy and increase the lifetime of the system by trading increased computation for reduced radio communication (which is relatively expensive in terms of energy cost [17]). Rather than collect samples centrally, it is generally desirable to perform local compression, aggregation, or summarization within the sensor network to reduce overall communication overheads. As a simple example, consider determining the boundary of a region of interest in the network. While this is straightforward to implement centrally if all sensor data is known, a localized version would operate by exchanging data between nearby sensors. Communicating only the boundary threshold to the base station yields significant communication and energy savings.

The core difficulty with building sensor network applications is that high-level, global behavior must be expressed in terms of complex, local actions taken at each node. In the boundary finding example, instead of running standard edge-detection algorithms on the complete set of sensor data, it is necessary to implement techniques that rely chiefly on local data and limited communication between nearby nodes. This challenge is amplified by the limitations of sensor hardware, making traditional parallel algorithms unattractive: such decomposition generally assumes the existence of efficient, any-to-any communication between processors.

In contrast, sensor applications focus on local communication within the network, allowing nodes to coordinate activity with few radio messages. For example, if nodes are able to perform boundary detection by communicating only with other nodes within single-hop (broadcast) radio range, no routing is required, simplifying protocol design and saving energy. This form of local, spatial processing is a powerful paradigm for in-network processing, and is found in a number of existing applications. Currently, developers are required to build up their own mechanisms for such local coordination. Our goal is to simplify application design by identifying a set of reusable, general-purpose communication primitives based on local regions.

Given such extreme resource limitations, it is critical that any communication abstraction for sensor networks allow the application some measure of control over resource consumption. Unlike more traditional environments, where the primary goal is performance, managing communication overheads is essential for meeting energy and bandwidth goals. Moreover, given the reactive nature of sensor networks, there is the opportunity to design applications that can adapt to changing network or environmental conditions, tuning the bandwidth or energy usage of the communication model to achieve given targets of latency, accuracy, or lifetime.

In this paper, we describe *abstract regions*, a family of spatial operators that capture local communication within regions of the network, which may be defined in terms of radio connectivity, geographic location, or other properties of nodes. In addition to providing a flexible means of node addressing, abstract regions support data sharing using a tuple-space-like programming model, as well as efficient aggregation operations. In addition, abstract regions expose the tradeoff between the accuracy and resource usage of communication operations. Applications can adapt to changing network conditions by tuning the energy and bandwidth usage of the underlying communication substrate. Abstract regions provide feedback on the accuracy of collective operations as well as an interface for controlling resource usage. An earlier position paper [33] introduced the abstract regions interface and its use for tuning energy/accuracy tradeoffs. In this paper, we expand on these ideas and present a detailed study of their applications and performance.

Abstract regions are general enough to support a wide range of sensor network applications and form the basis for other, higher-level communication models. In this paper, we describe the abstract regions programming model, as well as several implementations, including regions based on geographic and radio neighborhoods, spanning trees, and an approximate planar mesh. We have implemented abstract regions in nesC [11], a component-oriented variant of C used by the TinyOS [17] operating system. To further simplify abstract region programming, we have implemented a lightweight, thread-like abstraction for TinyOS, allowing applications to invoke blocking operations, which were heretofore disallowed by the TinyOS concurrency model.

We present four applications based on the abstract region model: tracking a moving object through a sensor field, detecting a contour in sensor readings, event detection based on a variant of directed diffusion [18], and geographic routing using GPSR [19]. We evaluate the accuracy and communication overhead of each of these applications and present results highlighting the tradeoff between resource consumption and quality of the results. The rest of this paper is organized as follows. Section 2 motivates our approach and discusses related work. Section 3 introduces the abstract region programming model and the use of resource tuning. Section 4 describes the implementation of abstract regions in the TinyOS operating system. Section 5 presents three applications based on regions, and Section 6 evaluates the accuracy and resource consumption tradeoff provided by the regions programming model. Section 7 discusses future work and concludes.

2 Motivation and Background

Sensor networks have attracted increasing interest from research and industry. The potential to instrument the physical world at high resolution and low cost opens up a wide range of novel applications in areas such as engineering [7, 20], biology [8, 27], and medicine [9, 34]. The future success of sensor networks depends to a large extent on the programming and communication abstractions presented to application developers. This is especially critical when one considers that the primary developers for sensor networks will be scientists and engineers. These users require a programming model that closely represents the high-level, data-centric computation to be performed within the network, rather than the existing, low-level, node-centric programming interfaces provided by existing systems.

In this paper, we focus on sensor networks based on small, low-power sensors such as the UC Berkeley MICA node. This device consists of a 7.3 MHz ATmega128L processor, 128KB of code memory, 4KB of data memory, and a Chipcon CC1000 radio capable of 38.4 Kbps and an outdoor transmission range of approximately 300m. The device measures 5.7cm $\times$ 3.1cm $\times$ 1.8cm and is typically powered by 2 AA batteries with an expected lifetime of days to months, depending on application duty cycle. The limited memory and computational resources of this platform make an interesting design point, as software layers must be tailored for this restrictive environment. The MICA node uses a lean, component-oriented operating system, called TinyOS [17], and an unreliable message-passing communication model based on Active Messages [32].

2.1 The need for abstractions

The bandwidth and energy limitations of sensor nodes typically require that in-network processing be performed to reduce the amount of data that must be transferred out of the network. Application designers are therefore faced with the problem of decomposing an initially straightforward data-collection task into a parallel program with local communication among sensor nodes. Currently, sensor application designers spend a great deal of effort building up low-level machinery for routing, data collection, and energy management. We wish to move away from this state of affairs by providing higher-level programming interfaces that abstract these details, yet provide enough flexibility to implement efficient algorithms. The first step in this endeavor is to identify an appropriate set of programming primitives, and is the primary focus of this paper.

Our approach can be likened to that of MPI [13], which provides a unified interface for message passing across a large family of parallel machines. MPI hides the details of the communication hardware and provides efficient implementations of common collective operations, such as broadcast and reduction. MPI has been extremely successful in the parallel processing community as it is high-level enough to shield programmers from most of the details of the underlying machine, yet low-level enough to permit extensive application-specific optimizations. We wish to provide communication interfaces that serve a similar role for sensor networks.

2.2 Related work

Several communication and programming models for sensor networks have been proposed in the literature. Among the first was directed diffusion [18], which provides a framework for distributed event detection and propagation. This approach has been extended by Heidemann *et al.* [14] to support in-network filtering and aggregation. In contrast, our focus is on a lower-level mechanism, namely, maintaining and coordinating regions of nodes for in-network processing. Later in this paper, we show that abstract regions can be used to implement a form of directed diffusion (see Section 5.3). Other communication abstractions include GHT [30], Spatial Programming [3], DIFS [12], SPIN [15], and DIMENSIONS [10]. These systems are generally focused on a specific communication or aggregation model rather than supporting a wide range of applications.

Most closely related to our approach is Hood [35], a neighborhood-based programming abstraction for sensor networks. Like abstract regions, Hood provides a mechanism for discovery and sharing of data among sensor nodes. The shared variable mechanism described in Section 3 was inspired by the Hood model, although Hood provides a richer set of primitives for manipulating shared data. Unlike abstract regions, Hood does not directly address the resource/accuracy tradeoff or provide a mechanism for data aggregation.

TinyDB [25], Cougar [38], and IrisNet [28] provide a high-level SQL or XML-based query interface to sensor network data. Queries are deployed into the network, streaming results to one or more base stations, and aggregation is used to reduce communication overhead. These systems have tremendous value and abstract away many details of communication, aggregation, and filtering. By the same token, however, they are not well-suited for developers who wish to implement specific behavior at a lower level than the query interface.

For example, TinyDB is focused on relaying aggregate data along a spanning tree rooted at a base station. While this mechanism can support complex algorithms such as contour finding, TinyDB must expose an internal "contour finding operator" to queries [16]. Our concern is with the implementation of these lower-level operators themselves, and with providing a set of communication abstractions that can be used to implement higher-level services, such as queries.

2.3 Resource management and adaptation

It is critical that communication abstractions for sensor networks take resource management into account, and give applications control over the tradeoff between resource usage and the accuracy and yield of collective operations [33]. The energy usage of communication is dictated by a number of factors, including hardware properties, node density, radio channel quality, and local activity within the network. Moreover, these factors are generally not known *a priori* and may be highly dynamic. As a result, it is often desirable to consume fewer resources to obtain an approximate result, rather than pay an arbitrary resource cost for complete accuracy.

Adlakha *et al.* [1] describe a design-time recipe for tuning aspects of sensor networks to achieve given accuracy, latency, or lifetime goals. However, this approach assumes a statically-configured network where resource requirements are known in advance, rather than allowing the network behavior to be tuned at runtime (say, in response to increased activity).

Several adaptive communication mechanisms for sensor networks have been proposed. SPIN [15] is a set of data dissemination protocols that adapt to energy availability by reducing protocol overhead when energy resources are low. TinyDB provides a *lifetime* keyword that scales the query sampling and transmission period to meet a user-supplied network lifetime [26]. Boulis *et al.* [4] describe an aggregation mechanism that exposes an energy/accuracy knob to the user. These approaches are a step in the right direction, and our goal is to generalize the communication abstraction exposed to applications while retaining this approach of yielding control over resource usage.

3 Abstract Regions

Sensor network applications are often expressed in terms of groups of nodes over which local sampling, computation, and communication occur. For example, tracking a moving object involves aggregating sensor readings from nodes near the object. Abstract regions are a communication abstraction intended to simplify application development by providing a region-based collective communication interface. Abstract regions capture the inherent locality of communication and hide the details of data dissemination and aggregation within regions.

An abstract region defines a *neighborhood relationship* between a particular node and other nodes in the network. Examples of neighborhood predicates include "the set of nodes within N radio hops" and "the set of nodes within distance d." Local, spatial operations in the sensor network are performed by sharing data and coordinating activity among nodes in this neighborhood. In general, each node in the sensor network defines multiple abstract regions that it wishes to operate over, depending on application requirements. A node may also be a member of multiple regions at once: for example, a node will belong to multiple single-hop radio regions, one for each of its own set of single-hop neighbors.

3.1 Programming Abstraction

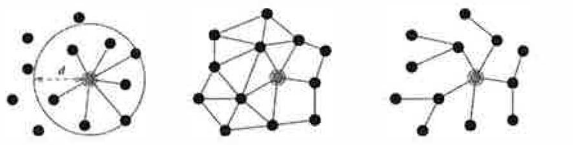
Regions capture a range of common idioms in sensor network programming. These include identification of neighboring nodes, data sharing, and data reduction within local neighborhoods. These operators allow nodes to query the state of neighboring nodes and implement efficient aggregation, compression, and summarization of local data. This programming model is based on that originally proposed by Hood [35]. Abstract regions support the following set of operators:

Neighbor discovery: Before performing other operations on a region, each node initiates the process of discovering neighboring nodes. Depending on the type of region, this may require broadcasting messages, collecting information on node locations, or estimating radio link quality. This is a continuous process, and each node is informed of changes in the region membership set, due to nodes joining, leaving, or moving within the network. A node may terminate this process at any time to avoid additional discovery messages. When terminated, the neighbor discovery operator returns a quality metric that measures the accuracy of the region formation, such as the percentage of candidate nodes that responded to the discovery request.

Enumeration: The enumeration operator returns the set of nodes participating in the region, allowing them to be addressed, for example, for direct message communication. Supplemental information, such as the location of each node, may be returned as well.

Data sharing: The data sharing operator allows variables, represented as key/value pairs, to be shared among nodes in the region. A shared variable supports two operations: *get* and *put*. *get(v,n)* retrieves the value of variable v from node n, and *put(v,l)* stores the value l of variable v at the local node. A simple implementation of *get(v,n)* sends a message to node n requesting the value of v. In this case, *put(v,l)* simply stores the value of v locally. An alternate implementation might broadcast or gossip the values of shared variables among nodes in the region, allowing *get(v,n)* to fetch locally cached values.

Reduction: The reduction operator takes a shared variable key and an associative operator (such as *sum*, *max*, or *min*) and reduces the shared variable across nodes in the region, storing the result in a shared variable. Reduction may be implemented in a number of ways, such as by collecting all values locally, or forming a spanning tree and propagating values up the branches of the tree, performing reduction at each level. As with shared variables, the region hides the details of reductions from the programmer.



(a) Geographic (b) Planar mesh (c) Spanning tree

Figure 1: **Examples of abstract regions.**

Abstract regions simplify application design by shielding developers from the complexity of routing, data dissemination, and state management. Additionally, regions provide a unified interface regardless of the particular definition of the region membership. That is, one can readily interchange the underlying region implementation without necessarily affecting higher-level application logic. For example, an application that makes use of an N-radio-hop region can be readily modified to use a geographic neighborhood region in its place.

3.2 Abstract Region Implementations

Given the diverse needs of sensor network applications, we expect a range of abstract region definitions will be useful to programmers. We have completed several abstract region implementations, with several others underway. Three examples are shown in Figure 1. They include:

- *N-radio hop:* Nodes within N radio hops;
- *N-radio hop with geographic filter:* Nodes within N radio hops and distance d;
- *k-nearest neighbor:* k nearest nodes within N radio hops;
- *k-best neighbor:* k nodes within N radio hops with the highest link quality, as measured in fraction of packets dropped over some measurement interval;
- *Approximate planar mesh:* A mesh with a small number (possibly zero) crossing edges; and
- *Spanning tree:* A spanning tree rooted at a single node, used for aggregating values over the entire network.

3.2.1 Radio and geographic neighborhoods

The implementation of the radio and geographic neighborhoods is straightforward. To discover neighbors, each node broadcasts periodic advertisements. Data sharing may be implemented using either a "push" (*put* broadcasts updates to neighboring nodes) or "pull" (*get* sends a fetch message to the corresponding node) approach;

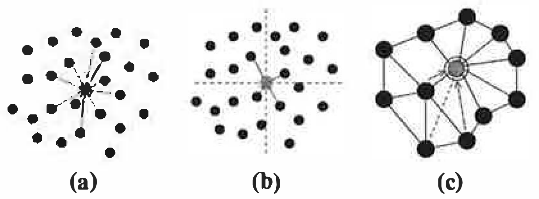


Figure 2: **Forming an approximate planar mesh region.** *(a) Nodes first create a region of k-nearest neighbors. (b) The nearest neighbor in each sector is chosen as an outedge and advertised. (c) If an advertised edge crosses another edge, invalidation messages are sent to the source.*

our current implementation uses the latter. Reduction involves collecting shared variable values locally, combining them with the reduction operator, and storing the result in another shared variable.

3.2.2 *Approximate planar mesh*

Planar meshes, such as the Delaunay triangulation [31], are useful for spatial computation (such as dividing space into nonoverlapping cells) and geographic-based routing [19]. However, algorithms for constructing planar meshes generally require either global knowledge of the connectivity graph or extensive interprocessor communication. We have implemented an *approximate* planar mesh in which a small number of edges may cross, but which uses communication only within single-hop neighborhoods of each node.

Our algorithm is based on a pruned Yao graph [23] and is depicted in Figure 2. First, each node discovers a k-nearest radio region of candidate nodes. When k neighbors have been accumulated, each node forms the Yao graph by dividing space around it into m equal-sized sectors of angle $\theta = 2\pi/m$ and selecting the nearest node within each sector as a potential neighbor. Next, each node uses a single-hop broadcast to advertise its selected outedges. Upon reception of an edge advertisement, each node tests whether the given edge crosses one of its own outedges, and if so sends an invalidation message to the source, causing it to prune the offending edge from its set. Nodes do not select additional neighbors beyond the initial candidates, so a node may end up with fewer than m outedges. Nodes perform several rounds of edge set broadcasts and wait for some time before settling on a final set of neighbors. Both the number of broadcasts and the timeout settings can be tuned by the application, as discussed below in Section 3.3. Changes in the underlying k-nearest-neighbor set initiates selection of new Yao neighbors. Data sharing and reduction are implemented using the same components as in the radio neighborhood, as all neighbors are one radio hop away.

We have also implemented planar regions based on the Relative Neighborhood Graph and Gabriel Graph, and an implementation of GPSR [19] based on them. This is described in Section 5.4.

3.2.3 *Spanning tree*

Spanning trees are useful for aggregating values within the sensor network at a single point, as demonstrated by systems such as TinyDB [25] and directed diffusion [18]. Our spanning tree implementation provides the same programmatic interface as other regions, although the semantics of the shared variable and reduction operations have been augmented to support *global* communication through the routing topology provided by the spanning tree. The shared variable *put* operation at the root floods the shared value to all nodes in the tree, while a *put* at a non-root node propagates the value to the root. Reduction operations cause data to propagate up the tree, causing each node to aggregate its local value with that of its children.

The spanning tree implementation adapts to changing network conditions, as nodes attempt to select a parent in the spanning tree to maximize the link quality and minimize the message hopcount to the root. Each node maintains an estimate of the link quality to its parent, measured in terms of the fraction of successful message transmissions to the parent and the fraction of messages successfully received from other nodes, using a simple sequence number counting scheme.

The tree is formed by broadcasting messages indicating the source node ID and number of hops from the root; nodes rebroadcast received advertisements with their own ID as the source and the hopcount incremented by 1. Nodes initially select a parent in the tree based on the lowest hopcount advertisement they receive, but may select another parent if the link quality estimate falls below a certain threshold. This estimate is smoothed with an exponentially weighted moving average (EWMA) to avoid rapid parent reselections. Our goal in this work has not been to develop the most robust tree formation algorithm, but rather to demonstrate that spanning trees fit within the programming interface provided by abstract regions. It would be straightforward to replace the tree construction algorithm with another, such as that described in [36], and layer the abstract regions interface over it.

3.3 Quality feedback and tuning interface

The collective operations provided by abstract regions are inherently unreliable. The quality of region discovery, the fraction of nodes contacted during a reduction operation, and the reliability of shared variable operations all depend on the number of messages used to perform those operations. Generally, increasing the number of messages (and hence, energy usage) improves communication reliability and the accuracy of collective operations. However, given a limited energy or bandwidth budget, the application may wish to perform region operations with reduced fidelity.

Abstract regions expose this tradeoff between resource consumption and the accuracy or yield of collective operations. This is in contrast to most existing approaches to sensor network communication, which hide these resource tradeoffs from the application. As a concrete example of such a tradeoff, the communication layer may tune the number and frequency of message retransmissions to obtain a given degree of reliability from the underlying radio channel. Similarly, nodes may vary the rate at which they broadcast location advertisements to neighboring nodes, affecting the quality of region discovery.

Abstract regions provide feedback to the application in the form of a *quality measure* that represents the completeness or accuracy of a given operation. For member discovery, the quality measure represents the fraction of candidate nodes that responded to the discovery request. For example, when discovering one-hop neighbors within distance *d*, the quality measure represents the fraction of one-hop nodes that responded to the region formation request. For reduction, the quality measure represents the fraction of nodes in the region that participated in the reduction. In addition, each operation supports a timeout mechanism, in which the operation will fail if it has not completed within a given time interval. For example, when performing a shared variable *get* operation, a timeout indicates that the data could not be retrieved from the requested node.

Applications can use this quality feedback to affect resource consumption of collective operations through a *tuning interface*. The tuning interface allows the application to specify low-level parameters of the region implementation, such as the number of message broadcasts, amount of time, or number of candidate nodes to consider when forming a region. Likewise, region operations perform message retransmission and acknowledgment to increase the reliability of communication; the depth of the transmit queue and number of retransmission attempts can be tuned by the application.

In our current implementation, the set of parameters exposed by the tuning interface is somewhat low-level and in many cases specific to the particular region implementation. Many of these parameters are straightforward to tune, although their effect on resource consumption may not be immediately evident, as it is highly dependent on application behavior. In general, an application designer will need to study the impact of the relevant tuning parameters on resource usage and application performance.

Exposing tuning knobs and quality feedback greatly impacts the programming model supported by abstract regions. Rather than hiding performance tradeoffs within a generic communication library with no knowledge of application requirements, the tuning interface enables applications to adapt to changing environments, network conditions, node failure, and energy budget. For example, this interface can be used to implement adaptive control mechanisms that automatically tune region parameters to meet some quality or resource usage target. In Section 6, we demonstrate an adaptive controller that tunes the maximum retransmission count to achieve a target yield for reduction operations.

We are currently working on a resource-centric tuning mechanism that allows the programmer to express an energy, radio bandwidth, or latency budget for each operator, and which maps these constraints onto the appropriate low-level parameter settings. This approach shields the programmer from details of the low-level interface and enables adaptivity using higher-level resource constraints. This is discussed further in Section 7.

4 Implementation

In this section we detail the implementation and programming interface for abstract regions on TinyOS [17], a small operating system for sensor nodes. Unlike the event-driven, asynchronous concurrency model provided by TinyOS, abstract regions provide a blocking, synchronous interface, which greatly simplifies application logic. This is accomplished using a lightweight, thread-like abstraction called *fibers*, described below.

4.1 TinyOS concurrency model

Traditional concurrency mechanisms, such as threads, are too heavyweight to implement on extremely resource-constrained devices such as motes. TinyOS employs an event-driven concurrency model, in which long-running operations are not permitted to block the application, but rather using a *split-phase* approach. An initial request for service is invoked through a *command*, and when the operation is complete, an *event* (or callback) is invoked on the original requesting component. For example, to take a sensor reading, an application invokes the `getData()` command on the sensor hardware component, which later invokes the `dataReady()` callback supplying the sensor reading.

The TinyOS concurrency model requires each concurrent "execution context" to be implemented manually by the programmer as a state machine, with execution driven by the sequence of commands and events invoked on each software component. If an application is performing multiple concurrent tasks, these operations must be carefully interleaved. In addition, each split-phase operation requires that the application code be broken across multiple disjoint segments of code. The programmer must manually maintain continuation state across each split-phase operation, adding significant complexity to the code. While the logical program may be quite simple, the lack of blocking operations in TinyOS requires that the application be broken into multiple tasks and event handlers.

A key observation is that a broad class of sensor network applications can be represented by a small number of concurrent activities: for example, a core application loop that periodically performs some sensing and communication operations, as well as a reactive context that responds to external events (e.g., incoming messages). Providing blocking operations in TinyOS does not require a full-fledged threads mechanism, but rather the means to support a limited range of blocking operations along with event-driven code to handle asynchronous events.

4.2 Lightweight Fibers

We have implemented a lightweight, thread-like concurrency model for TinyOS, supporting a single blocking execution context alongside the default, event-driven context provided by TinyOS. We use the term *fibers* to refer to each execution context.[1] The default *system fiber* is event-driven and may not block, while the *application fiber* is permitted to block. Because there is only one blocking context, both the system and application fibers can share a single stack. The overhead for each fiber, therefore, consists of a saved register set, which on the ATmega128L is 24 bytes. The runtime overhead for a context switch is just 150 instructions. Apart from the single blocking context, applications may employ additional concurrency through TinyOS's event-driven concurrency model.

The application fiber may block by saving its register set and jumping to the system fiber. The system fiber restores its registers, but resumes execution on the application stack. An event, such as a timer interrupt or message arrival, may wake up the application fiber, causing its register state to be restored once the event handler has completed. The system fiber maintains no live state on the stack after the application fiber is resumed.

Fibers allow blocking and event-driven concurrency to be freely mixed in an application, and different components can use different concurrency models depending on their requirements. In general, the central sense-process-communication loop of an application may employ blocking for simplicity, while the bulk of the TinyOS code remains event-driven.

4.3 Abstract region API

The abstract regions programming interface is shown in Figure 3. The use of blocking interfaces, provided by fibers, greatly simplifies application design. Rather than breaking application logic across a set of disparate event handlers, a single, straight-line loop can be written. Apart from considerably shortening the code, this approach avoids common programming errors, such as maintaining continuation state incorrectly.

Without blocking calls, the object tracking application described in Section 5.1 is 369 lines of nesC code, consisting of 5 event handlers and 11 continuation functions. The corresponding blocking version (shown in pseudocode form in Section 5.1) is 134 lines of code and consists of a single main loop. All application state is stored in local variables within the loop, rather than being marshaled into and out of global continuations for each split-phase call.

[1]This term is borrowed from Windows, in which fibers are explicitly scheduled by the application [2]

```
/* Discover region */
result_t Region.formRegion(<region specific args>,
  int timeout);

/* Wait for region discovery */
result_t Region.sync(int timeout);

/* Set local shared variable */
result_t SharedVar.put(sv_key_t key, sv_value_t val);

/* Get shared variable from give node */
result_t SharedVar.get(sv_key_t key, addr_t node,
  sv_value_t *val, int timeout);

/* Wait for shared variable gets */
result_t SharedVar.sync(int timeout);

/* Reduce 'value' to 'result' with given op */
/* 'yield' returns pct of nodes responding */
result_t Reduce.reduceToOne(op_t operator,
  sv_key_t value, sv_key_t result,
  float *yield, int timeout);

/* Reduce and set result in all nodes */
result_t Reduce.reduceToAll(op_t operator,
  sv_key_t value, sv_key_t result,
  float *yield, int timeout);

/* Wait for reductions to complete */
result_t Reduce.sync(int timeout);
```

Figure 3: **Abstract regions programming interface.**

Abstract regions themselves are implemented using event-driven concurrency, due to the number of concurrent activities that the region must perform. For example, the spanning tree region is continually updating routing tables based on link quality estimates received from neighboring nodes; the reactive nature of this activity is better suited to an event-driven model. A wrapper component is used to provide a blocking, fiber-based interface to each abstract region implementation, and applications can decide whether to invoke a region through the blocking or the split-phase interface.

5 Applications

To demonstrate the effectiveness of regions, in this section we describe four sensor network applications that are greatly simplified by their use. These include tracking an object in the sensor field, finding spatial contours, distributed event detection using directed diffusion [18], and geographic routing using GPSR [19].

5.1 Object tracking

Object tracking is an oft-cited application for sensor networks [5, 22, 37]. In its simplest form, tracking involves determining the location of a moving object by detecting changes in some relevant sensor readings, such as magnetic field.

Our version of object tracking uses a simple algorithm based on the DARPA NEST demonstration software, described in [35]. Each node takes periodic magnetometer readings and compares them to a threshold value. Nodes above the threshold communicate with their neighbors and elect a leader node, which is the node with the largest magnetometer reading. The leader computes the centroid of its neighbors sensor readings, and transmits the result to a base station.

The following pseudocode shows this application expressed in terms of abstract regions:[2]

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region.create(8);

while (true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node ... */
    if (max_id == my_id) {
      /* Perform reductions and compute centroid */
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

The program performs essentially all communication through the abstract regions interface, in this case the k-nearest-neighbor region. Nodes store their local sensor reading and the reading scaled by the x and y dimensions of their location as shared variables. Nodes above the threshold perform a reduction to determine the node with the maximum sensor reading, which is responsible for calculating the centroid of its neighbors' readings. A series of sum-reductions is performed over the shared variables which are used to compute the centroid:

$$\begin{aligned} c_x &= \sum_i R_i x_i / \sum_i R_i \\ c_y &= \sum_i R_i y_i / \sum_i R_i \end{aligned}$$

where c_x and c_y are the (x, y)-coordinates of the centroid, R_i is the reading at node i, and x_i and y_i are the (x, y)-coordinates of node i.

The use of regions greatly simplifies application design. The programmer need not be concerned with the details of routing, data sharing, or identifying the appropriate set of neighbor nodes for performing reductions. As a result the application code is very concise. As described earlier, our actual implementation is just 134 lines of code, of which 70 lines comprise the main application loop above. The rest consists of variable and constant declarations as well as initialization code.

As a rough approximation of the linecount for a "hand coded" version of this application, without the benefit of the abstract regions interface, consider the combined size of the nonblocking object tracking code (369 lines) and the k-nearest-neighbor region code (247 lines). This total of 616 lines, compared with 134 lines for the abstract region version, suggests that this programming abstraction captures a great deal of the complexity of sensor applications.

[2]For brevity, the use of tuning and quality feedback is not shown in this example.

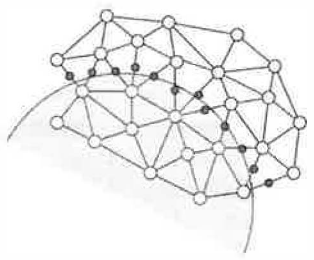


Figure 4: **Contour finding application.** *The shaded region represents an area where sensor readings fall above a threshold. Nodes (unfilled circles) are connected into an approximate planar mesh. Contour points (filled circles) are chosen as midpoints between a node above the threshold and a node below the threshold.*

5.2 Contour finding

Another typical sensor network application is detecting contours or edges in the set of sensor readings [16, 24]. Contour finding involves determining a set of points in space that lie along, or close to, an isoline in the gradient of sensor readings. Contours might represent the boundary of a region of interest, such as a group of sensors with an interesting range of readings. Contour finding is a valuable spatial operation as it compresses the per-node sensor data into a low-dimensional surface. This primitive could be used for detecting thermoclines or tracking the flow of contaminants through soil [6].

An implementation of contour finding using abstract regions is depicted in Figure 4, and is expressed in pseudocode as:

```
location = get_location();
/* Form approximate planar mesh */
region = apmesh_region.create();
region.putvar(loc_key, location);

while (true) {
  reading = get_sensor_reading();
  region.putvar(reading_key, reading);

  if (reading > threshold) {
    foreach (nbr in region.get_neighbors()) {
```

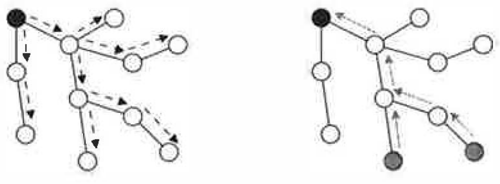

(a) (b)

Figure 5: **Directed diffusion operation.** *(a) The sink forms a spanning tree and floods interests throughout the network. (b) Matching nodes route event reports to the corresponding sink through the spanning tree topology.*

```
        /* Fetch neighbor's reading */
        rem_reading = region.getvar(reading_key, nbr);
        if (rem_reading <= threshold) {
          rem_loc = region.getvar(loc_key, nbr);
          contour_point = midpoint(location, rem_loc);
          send_to_basestation(contour_point);
        }
      }
    }
    sleep(periodic_delay);
  }
```

Nodes first form an approximate planar mesh region, as described in Section 3. Each node stores its location and sensor reading as a shared variable. Nodes that are above the sensor threshold of interest fetch the readings and locations of their neighbors. For each neighbor that is below the threshold, the node computes a contour point as the midpoint between itself and its neighbor, and sends the result to the base station.

The use of the approximate planar mesh region ensures that few edges will cross, and that nodes will generally select neighbors that are geographically near. These properties are vital for our contour finding algorithm as it is based on pairwise comparisons of sensor readings, and values are computed along edges in the mesh. As with the object tracking application, abstract regions shield the programmer from details of mesh formation and data sharing; the application code is very straightforward. It is only 118 lines of nesC code, 56 lines of which are devoted to the main application loop.

5.3 Directed diffusion

As discussed in Section 2, directed diffusion [18] has been proposed as a mechanism for distributed event detection in sensor networks. The idea is to flood *interests* for named data throughout the network (such as "all nodes with sensor readings greater than threshold T"), as shown in Figure 5. Nodes that match some interest report their local value to the *sink* that generated the interest. Here, we show that abstract regions can be used to implement the directed diffusion mechanism, demonstrating their value for building higher-level communication abstractions.

The directed diffusion programming interface consists of two operations: *broadcastInterest* and *publishData*. Sink nodes invoke *broadcastInterest* with an operator (such as $=$, $>$, or $<=$) and a comparison value. Our current implementation supports interests in the form of simple binary operator comparisons; it would be straightforward to extend this to support more general interest specifications. Nodes call *publishData* to provide local data, which is periodically compared against all received interests. When the local data matches an interest, the node routes an event report to the corresponding sink. When the sink receives an event report, a *dataReceived* event handler is invoked with the corresponding node address and value.

Our directed diffusion layer is implemented using the spanning tree abstract region. Data sinks first form an adaptive spanning tree, and publish an interest record by inserting it into the tuplespace associated with that region. Recall that in the spanning tree region, the tuplespace *put* operation, when invoked at the root, broadcasts the entry to all nodes in the tree. Nodes periodically check for interest records using the tuplespace *get* operation, and compare their current value (provided by `publishData()`) with all received interests. If the value matches, it is inserted into the tuplespace of the corresponding spanning tree, which propagates the value to the root.

Nearly all of the complexity of broadcasting interests and propagating data to the sink is captured by the spanning tree region and its tuplespace implementation. The directed diffusion layer is responsible only for representing interests and matching data values as tuplespace entries, allowing the spanning tree region to handle the details of maintaining routes between data sources and sinks. One drawback with our current implementation is that multiple sinks receiving the same data must form separate spanning trees, causing nodes with matching data to send multiple copies. However, it would be straightforward to optimize the spanning tree region by suppressing duplicate packets being sent along the same network path. The directed diffusion layer is only 188 lines of nesC code, of which 66 lines handle initialization and definitions. In contrast, the spanning tree region code is 937 lines.

5.4 Geographic routing using GPSR

Geographic routing protocols use the locations of nodes in the network to make packet forwarding decisions. Rather than maintaining distance vectors or link state at each node, a node only needs to know the geographic positions of its immediate neighbors. One such protocol is Greedy Perimeter Stateless Routing (GPSR) [19], which operates in two modes: *greedy routing*, in which the message is routed to the neighboring node that is closest to the destination, and *face routing*, which is used when no neighbor is closer to the destination than the current node. Face routing requires a planar mesh to ensure that these local minima can be traversed.

We have implemented GPSR using both our radio

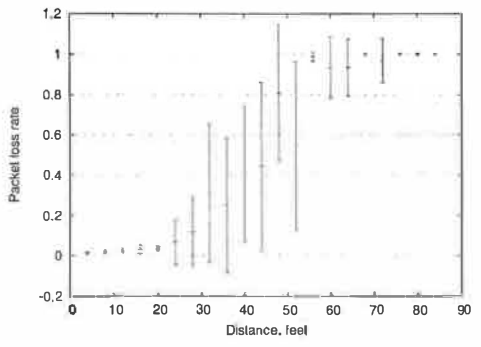


Figure 6: **TOSSIM radio loss model based on empirical data.** *The mean packet loss rate versus distance is shown, with errorbars indicating one standard deviation from the mean. The model is highly variable at intermediate distances.*

neighborhood region (for greedy routing mode) and approximate planar mesh regions (for face routing mode). The abstract regions code handles all aspects of constructing the appropriate graphs, greatly simplifying the GPSR code itself. There are three types of approxiate planar mesh: the pruned Yao graph (PYG), Relative Neighborhood Graph (RNG), and Gabriel Graph (GG). Each of these graphs has slightly different rules for determining inclusion of an edge using only local information about the immediate neighbors of each node. Our GPSR implementation provides a scalable *ad hoc* routing framework for sensor networks, although space limitations prevent us from presenting detailed results.

6 Evaluation

In this section, we evaluate the abstract region primitives and demonstrate their ability to support tuning of resource consumption to achieve targets of energy usage and reliability. We investigate five scenarios: adaptive shared variable reduction, construction of an approximate planar mesh, object tracking, contour finding, and event detection using directed diffusion. In each case, we explore the use of the tuning interface to adjust resource usage and evaluate its effect on the accuracy of region operations.

These results were obtained using TOSSIM [21], a simulation environment that executes TinyOS code directly; our abstract region code can either run directly on real sensor motes or in the TOSSIM environment. TOSSIM incorporates a realistic radio connectivity model based on data obtained from a trace gathered from Berkeley MICA motes in an outdoor setting, as shown in Figure 6. This loss probability captures transmitter interference during the original trace that yielded the model. More detailed measurements would be required to capture the full range of transmission characteristics, although experiments have shown the model to be highly accurate [21]. We simulate a network of 100 nodes distributed semi-irregularly in a 20x20 foot

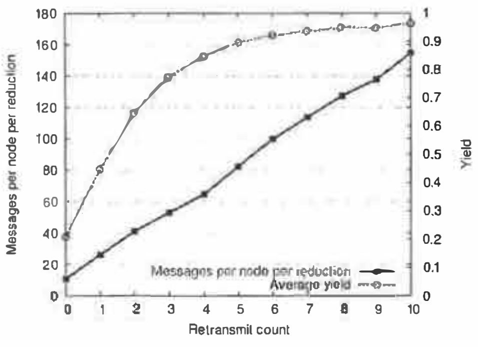


Figure 7: **Reduction yield and overhead in the k-nearest neighborhood region.** *This figure shows the average yield (fraction of neighbors responding to a reduction request) and number of messages for reduction operations. The yield and overhead are directly related to the number of retransmission attempts made by the transport layer.*

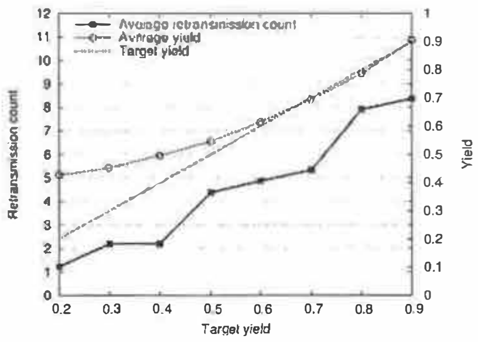


Figure 8: **Adaptive reduction algorithm performance.** *This figure shows the effectiveness of dynamically tuning the maximum retransmission count to meet a reduction yield target. The figure shows the average yield across 100 reduction operations, as well as the average retransmission count determined by the controller.*

area. Because TOSSIM does not currently simulate the energy consumption of nodes, we report the number of radio messages sent as a rough measure of energy consumption, which is reasonable given that on the MICA platform, radio communication dominates CPU energy usage by several orders of magnitude.

6.1 Adaptive reduction algorithm

By performing reduction over a subset of neighbors in a region, nodes can trade off energy consumption for accuracy. As described earlier, the reduction operator provides quality feedback in the form of the fraction of nodes that responded to the reduce operation. Over an unreliable radio link, this fraction is directly related to the number of retransmission attempts made by the underlying transport layer, as shown in Figure 7. Moreover, the appropriate retransmission count to meet a given yield target is a function of the local network density and channel characteristics, which vary both across the network and over time.

```
region = k_nearest_region.create(8);
region.putvar(key, local_id);
tuning.set(MAX_RETRANSMIT_COUNT, max_xmit);

while (true) {
  val = region.reduce(OP_MAX, key, &quality, timeout);
  avg_quality = (quality * ALPHA) +
    (avg_quality * (1.0 - ALPHA));

  if (avg_quality < (target - LOW_WATER)) {
    max_xmit++;
  } else if (avg_quality > (target + HIGH_WATER)) {
    if (--max_xmit < 1) max_xmit = 1;
  }
  tuning.set(MAX_RETRANSMIT_COUNT, max_xmit);
  sleep(delay);
}
```

Figure 9: **Pseudocode for adaptive reduction algorithm.**

We implemented a simple adaptive reduction algorithm that attempts to maintain a target reduction yield by dynamically adjusting the maximum number of retransmission attempts made by the transport layer. Pseudocode is shown in Figure 9, and illustrates the use of the regions tuning interface. Nodes form a k-nearest-neighbor region ($k = 8$) and repeatedly perform a max-reduce over a local sensor reading. An additive-increase/additive-decrease controller is used, which takes an exponentially-weighted moving average (EWMA) of the yield of each reduction operation. If the yield is 10% greater than the target, the maximum retransmission count is reduced by one; if it is 10% less than the target, the count is increased by one.

This simple algorithm is very effective at meeting a given yield target, as shown in Figure 8. Note that for targets below 0.5, the controller overshoots the target somewhat. This is mainly because most nodes are well-connected, and even a low retransmission count will result in a good fraction of messages getting through. Another interesting metric is the average number of messages exchanged for each reduction operation (not shown in the figure). Reductions are implemented by the root sending a request message to each neighbor in the region, which replies with the value of the requested shared variable. Therefore, with no message loss, two messages are exchanged per node. As the yield target scales, this number ranges from 3 (for a yield target of 0.2) to about 10.5 (for a target of 0.9), which gives an indication of the additional overhead for achieving a target reliability.

6.2 Approximate planar mesh construction

Constructing an approximate planar mesh is a tradeoff between the number of messages sent and the quality of the resulting mesh, which we measure in terms of the fraction of crossing edges. Given the unreliable nature of the communication channel, our pruned Yao graph algorithm cannot guarantee that the mesh will be planar. For many applications, a perfect mesh is not necessary, since planarity is impossible to guarantee if there is measurement error in node localization. As we will see in the next subsection, the quality of the planar mesh has a direct influence on the accuracy of contour detection.

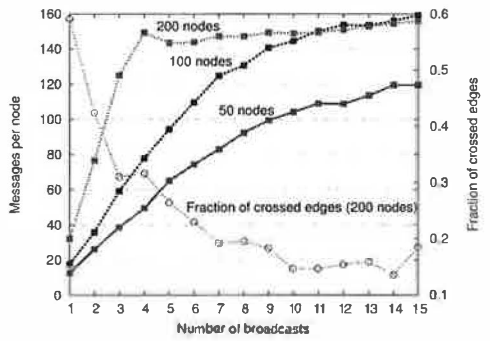


Figure 10: **Quality and overhead of the pruned Yao graph as a function of number of node advertisements.** *The fraction of crossed edges is substantially similar for 50 and 100 nodes.*

In our implementation, the fraction of crossed edges in the mesh depends primarily on the number of broadcasts made by each node to advertise its location. These advertisements are used to form the k-nearest-neighbor region, the first step in the planar mesh construction. Intuitively, hearing from a greater number of radio neighbors allows the k-nearest-neighbor region to select the closest neighbors from this set. Missing an advertisement may cause nodes to select more distant neighbors, leading to a larger proportion of crossed edges.

Figure 10 shows the cost in terms of the number of messages sent per node, as well as the fraction of crossed edges, as the number of node advertisements is increased. Note that the message overhead does not grow linearly with the broadcast count; this is because we are counting all messages involved in mesh construction, including exchanging location information, outedge advertisements, and edge invalidation messages. There is a clear relationship between increased communication, and hence increased bandwidth and energy usage, and the quality of the mesh. As in the case of reduction, applications can tune the number of broadcasts to meet a given target resource budget or mesh quality.

The effect of increasing network density is also shown in the figure. In all cases, nodes are distributed over a fixed geographic area. As density increases, so does the message overhead, since nodes have more neighbors to consider during graph construction.

6.3 Contour finding accuracy

Next, we evaluate the contour finding application, described in Section 5, in terms of the quality of the underlying approximate planar mesh. We characterize the accuracy of contour finding as the mean error between each computed contour point and the actual contour location. Crossed edges in the mesh are likely to introduce error in the contour calculation, as they generally connect two nodes that may not be closest to the actual contour.

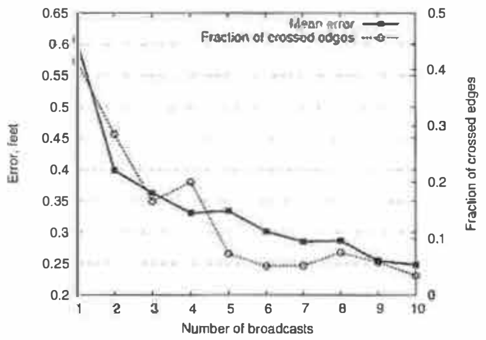


Figure 11: **Accuracy of contour detection tracking as a function of the number of node advertisements.**

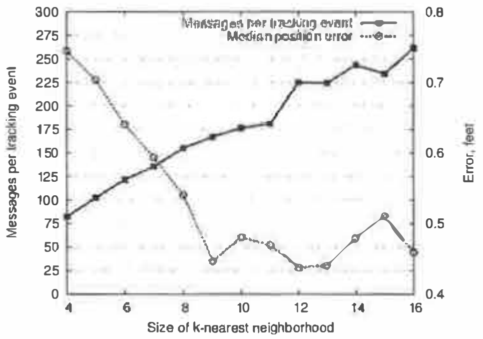


Figure 12: **Accuracy and overhead of object tracking as a function of neighborhood size.** *Results are the average of three runs for each neighborhood size.*

We simulated a linear contour passing through the center of the sensor field that rotates by an angle of 0.1 radians every 5 sec. Nodes take local sensor readings once a second and those that are above the sensor threshold compute new contour points. This scenario stresses the contour finding application and causes all areas of the network to eventually calculate contour points as the frontier rotates. In each case we ran the application for 100 simulated seconds.

Figure 11 shows the error in contour detection as the number of broadcasts used to construct the planar mesh is increased. There is a clear relationship between the overhead of mesh formation and the accuracy of contour detection. Also shown is the fraction of crossed edges, following a pattern similar to that in Figure 10.

6.4 Object tracking accuracy

Next, we evaluate the accuracy of the object tracking application described in Section 5. The application tracks a simulated object moving in a circular path of radius 6 feet at a rate of 0.6 feet every 2 sec. As with contour finding, moving the object in a circular path causes nodes in different regions of the network to detect and track the object. Nodes take sensor readings once a second, the value of which scales linearly with the node's distance to the object, with a maximum detection range of 5 feet. In each case we run the application for 100 simulated seconds and average over three runs.

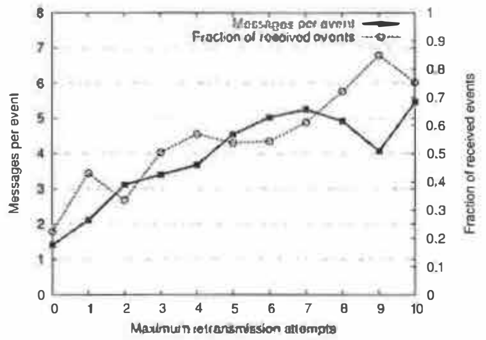


Figure 13: **Reliability and overhead of event detection as a function of retransmission attempts.**

The resource tuning parameter in question is the size of the k-nearest neighbor region. A smaller number of neighbors reduces communication requirements, but yields a less accurate estimate of the object location. Note that increasing the neighborhood size beyond a certain point will not increase tracking accuracy, as more distant nodes are less likely to respond to the reduction request due to packet loss.

Figure 12 shows the accuracy of the tracking application as we vary the number of neighbors in each region. For each time step, we calculate the distance between the simulated object and the value reported by the sensor network. As the figure shows, as the size of the neighborhood increases, so does the accuracy, as well as the number of messages sent (network-wide) per tracking event. Above about $k = 8$, the increase in the neighborhood size does not improve performance appreciably, in part because the additional nodes are further from the target object and may not detect its presence. Also, the more distant nodes may have poor radio connectivity to the root, as described above.

6.5 Event detection reliability

Finally, we evaluate the use of directed diffusion, implemented using the spanning tree region, to detect the presence of a moving object through the sensor field. The base station, located in the upper-left corner of the sensor field, forms a spanning tree region and floods interests to the network through the directed diffusion interface. Rather than computing the location of the object, nodes with a local sensor reading over a fixed threshold (corresponding to an object distance of 2.5 feet) route an event report back to the base. We compute the total number of messages as well as the fraction of event reports received at the base station as the number of retransmission attempts is scaled.

Figure 13 presents the results in terms of the number of messages transmitted per detected event, as well as the fraction of events received by the base station. As with the other examples, scaling the maximum retransmission count increases the number of messages, and has a corresponding effect on the reliability if event delivery to the base. In all of these cases, the core benefit of the abstract regions interface is the ability to tune the underlying communication substrate to achieve resource management and accuracy goals.

7 Conclusions and Future Work

As sensor networks become more common, better tools are needed to aid the development of applications for this challenging domain. We believe that programmers should be shielded from the details of low-level radio communication, addressing, and sharing of data within the sensor network. At the same time, the communication abstraction should yield control over resource usage and make it possible for applications to balance the tradeoff between energy/bandwidth consumption and the accuracy of collective operations.

The abstract region is a fairly general primitive that captures a wide range of communication patterns within sensor networks. The notion of communicating within, and computing across, a neighborhood (for a range of definitions of "neighborhood") is a useful concept for sensor applications. Similar concepts are evident in other communication models for sensor networks, although often exposed at a much higher level of abstraction. For example, directed diffusion [18] and TinyDB [25] embody similar concepts but lump them together with additional semantics. Abstract regions are fairly low-level and are intended to serve as building blocks for these higher-level systems.

We have described several abstract region implementations, including geographic and radio neighborhoods, an approximate planar mesh, and spanning trees. The implementation of abstract regions in the TinyOS environment relies on fibers, a lightweight concurrency primitive that greatly simplifies application design. Finally, we have evaluated the use of abstract regions for four typical sensor network applications: object tracking, contour finding, distributed event detection, and geographic routing. Our results show that abstract regions are able to provide flexible control over resource consumption to meet a given latency, accuracy, or energy budget.

In the future, we intend to explore how far the abstract region concept addresses the needs of sensor network applications. We are currently completing a suite of abstract region implementations and are developing several applications based on them. We also intend to provide a set of tools that allow application designers to understand the resource consumption and quality tradeoffs provided by abstract regions. These tools will provide developers with a view of energy consumption, communication overheads, and accuracy for a given application. Our goal is to allow designers to express tolerances (say, in terms of a resource budget or quality threshold) that map onto the tuning knobs offered by abstract regions. This process cannot be performed entirely off-line, as resource requirements depend on activity within the network (such as the number of nodes detecting an event). Runtime feedback between the application and the underlying abstract region primitives will continue to be necessary.

Our eventual goal is to use abstract regions as a building block for a high-level programming language for sensor networks. The essential idea is to capture communication patterns, locality, and resource tradeoffs in a high-level language that compiles down to the detailed behavior of individual nodes. Shielding programmers from the details of message routing, in-network aggregation, and achieving a given fidelity under a fixed resource budget should greatly simplify application development for this new domain.

Acknowledgements

The authors wish to thank Ion Stoica for his excellent advice in shepherding this paper, as well as the comments by the anonymous reviewers. Nelson Lee and Phil Levis provided much of the simulation framework used in the analyses. We also wish to thank Kamin Whitehouse and Cory Sharp for their comments on the neighborhood abstraction, as well as the tracking application upon which our algorithm is based.

References

[1] S. Adlakha, S. Ganeriwal, C. Schurgers, and M. B. Srivastava. Density, accuracy, latency and lifetime tradeoffs in wireless sensor networks - a multidimensional design perspective. In review, 2003.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management, or, event-driven programming is not the opposite of threaded programming. In *Proc. the USENIX 2002 Annual Conference*, June 2002.

[3] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proc. the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[4] A. Boulis, S. Ganeriwal, and M. B. Srivastava. Aggregation in sensor networks: An energy - accuracy tradeoff. In *Proc. IEEE workshop on Sensor Network Protocols and Applications*, 2003.

[5] R. Brooks, P. Ramanathan, and A. Sayeed. Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE*, November 2003.

[6] Center for Embedded Network Sensing. Contaminant transport monitoring. `http://cens.ucla.edu/Research/Applications/ctm.htm`.

[7] Center for Information Technology Research in the Interest of Society. Smart buildings admit their faults. `http://www.citris.berkeley.edu/applications/disaster_response/smartbuil%dings.html`, 2002.

[8] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proc. the Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.

[9] R. X. Cringely. Chase Cringely: Finding Meaning in a Lost Life. `http://www.pbs.org/cringely/pulpit/pulpit20020425.html`.

[10] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution search and storage in resource-constrained sensor networks. In *Proc. the First*

ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), November 2003.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Programming Language Design and Implementation (PLDI)*, June 2003.

[12] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A distributed index for features in sensor networks. In *Proc. the First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, Massachusetts, 1994.

[14] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proc. the 18th SOSP*, Banff, Canada, October 2001.

[15] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[16] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proc. the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.

[18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, Aug. 2000.

[19] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, Boston, MA, August 2000.

[20] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. In *Proc. the SPIE 10th Annual International Symposium on Smart Structures and Materials*, San Diego, CA, March 2000.

[21] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[22] D. Li, K. Wong, Y. H. Hu, and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), March 2002.

[23] X.-Y. Li, P.-J. Wan, Y. Wang, and O. Frieder. Sparse power efficient topology for wireless networks. In *Proc. 35th Annual Hawaii International Conference on System Sciences*, January 2002.

[24] J. Liu, P. Cheung, L. Guibas, and F. Zhao. A dual-space approach to tracking and sensor management in wireless sensor networks. In *Proc. the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[25] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[26] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. the ACM SIGMOD 2003 Conference*, June 2003.

[27] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.

[28] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[29] K. S. Pister. Tracking vehicles with a uav-delivered sensor network. `http://robotics.eecs.berkeley.edu/~pister/29Palms0103/`, March 2001.

[30] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage in sensornets. In *Proc. the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Atlanta, Georgia, September 2002.

[31] J. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-3):21–74, May 2002.

[32] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proc. the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[33] M. Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. In *Proc. the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[34] M. Welsh, D. Myung, M. Gaynor, and S. Moulton. Resuscitation monitoring with a wireless sensor network. In *Supplement to Circulation: Journal of the American Heart Association*, October 28, 2003.

[35] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[36] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[37] Y. Xu and W.-C. Lee. On localized prediction for power efficient object tracking in sensor networks. In *Proc. 1st International Workshop on Mobile Distributed Computing*, May 2003.

[38] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.

Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP

Jeffrey C. Mogul
HP Labs
Palo Alto, CA 94304
JeffMogul@acm.org

Yee Man Chan
Stanford Human Genome Center
Palo Alto, CA 94304
ymc@shgc.stanford.edu

Terence Kelly
HP Labs
Palo Alto, CA 94304
terence.p.kelly@hp.com

Abstract

Organizations use Web caches to avoid transferring the same data twice over the same path. Numerous studies have shown that forward proxy caches, in practice, incur miss rates of at least 50%. Traditional Web caches rely on the reuse of responses for given URLs. Previous analyses of real-world traces have revealed a complex relationship between URLs and reply payloads, and have shown that this complexity frequently causes redundant transfers to caches. For example, redundant transfers may result if a payload is *aliased* (accessed via different URLs), or if a resource *rotates* (alternates between different values), or if HTTP's cache revalidation mechanisms are not fully exploited. We implement and evaluate a technique known in the literature as *Duplicate Transfer Detection* (DTD), with which a Web cache can use digests to detect and potentially eliminate all redundant payload transfers. We show how HTTP can support DTD with few or no protocol changes, and how a DTD-enabled proxy cache can interoperate with unmodified existing origin servers and browsers, thereby permitting incremental deployment. We present both simulated and experimental results that quantify the benefits of DTD.

1 Introduction

Web caches are widely used to save bandwidth and improve latency. However, numerous studies have shown that, in practice, forward proxy caches (i.e., shared Web caches used near clients) incur miss rates of 51-70%, and byte-weighted miss rates of 64-86% [27,40]. Even warm caches with infinite storage cannot eliminate all misses.

In this paper, we are specifically concerned with redundant payload transfers, i.e., cases where a payload is transmitted to a recipient that has previously received it. In a traditional Web cache, each cache entry is indexed by a given URL. If a subsequent request arrives for that URL, and the cache cannot satisfy the request (it "misses"), it forwards the request to the origin server, which normally generates a reply containing a payload (Section 4.2 gives a careful definition for "payload"). If that exact payload has previously been received by the cache, we define this as a redundant payload transfer.

Others have identified the problem of redundant payload transfers on the World Wide Web, quantified its prevalence, and explored a range of possible solutions [2, 16,28]. According to one measurement, over 20% of payload transfers between origin servers and proxies are redundant [16].

We do not know all causes of redundant transfers. Many result from three common phenomena: *aliasing*, in which the same content is referenced under two different URLs; *rotation*, in which the same content is referenced twice under a single URL, but an intervening reference to that URL resolves to different content; and absent or faulty metadata that causes avoidable revalidation failures.

We previously proposed a technique called *Duplicate Transfer Detection* (DTD) [16] that allows any Web cache to potentially eliminate all redundant payload transfers, regardless of cause. DTD uses message digests to detect redundant transfers before they occur. In its use of digests to detect duplication, DTD is similar to approaches developed for other contexts, e.g., router-to-router packet transfers [32] and file systems for low-bandwidth environments [23]. Unlike an alternative proposal for eliminating redundant HTTP transfers [28], DTD does not require soft state that scales with the number of clients and the size of responses.

In [16] we did not propose a concrete protocol design or describe an implementation of DTD, nor did we measure its impact on client latency. In this paper, we show how one can use standard HTTP, with few or no explicit protocol changes, to support DTD without relying on any additional semantics, naming mechanisms, validation mechanisms, or cooperation with or between origin servers. This allows a DTD-enabled cache to interoperate with *unmodified* existing origin servers and browsers, thereby permitting smooth, incremental deployment. We describe how to implement DTD in a Web cache, and report on experiments showing that it can accomplish its goal of completely eliminating redundant transfers. We quantify the benefits of DTD using both experimental measurements of our implementation, and simulation results.

The main contributions of this paper are a well-defined protocol specification for DTD, the design of a real implementation of DTD, and performance evaluations of DTD.

2 Why eliminate redundant transfers?

Our DTD proposal does not reduce the number of times an HTTP cache must contact an origin server; it only reduces the number of response bodies that must be transferred. What makes this worthwhile?

Eliminating redundant transfers can improve at least four metrics:

Bandwidth: Web caches are often deployed to reduce bandwidth requirements (over half of large companies surveyed in 1997 cited bandwidth savings as their motivation for Web caching [11]). Redundant transfers consume bandwidth and increase peak bandwidth requirements.

Latency: Eliminating a redundant transfer can save latency in two ways: *directly*, by making the result available sooner (i.e., without having to wait for the redundant transfer to finish), and *indirectly*, by reducing channel utilization and thereby reducing queueing delays for subsequent responses.

Per-byte charges: Network tariffs are often flat-rate, but not always. In particular, wireless-data tariffs range from a few dollars to tens of dollars per Mbyte [3]. Redundant transfers on such networks directly waste money.

Energy: Studies have shown that energy consumption for wireless (and hence portable) networking is at least somewhat dependent on the amount of data transferred [10]. Eliminating some redundant transfers might therefore improve battery life.

In our previous study, using two large real-world traces, we showed that roughly 20% of payload transfers between origin servers and proxy caches are redundant [16]. Therefore, a solution to the redundant-transfer problem could yield significant savings on some or all of the metrics listed above. In this paper, we concentrate on quantifying these improvements.

3 Related work

The first published suggestion to eliminate redundant HTTP payload transfers using message digests, and a trace-based evaluation of its impact on Web cache hit rates, appeared in [15]. A recent unpublished undergraduate dissertation [4] develops a similar idea for GPRS Web access.

Santos & Wetherall [32] and Spring & Wetherall [33] describe protocol-independent network-level analogues of DTD that employ *packet* digests to save bandwidth. Muthitacharoen *et al.* designed a network file system for low-bandwidth environments that performs similar operations on chunks of files [23].

Web caches can use payload digests to avoid wasting *storage* as well as bandwidth. We have implemented this natural counterpart of DTD (see Section 8) but we are not the first. Bahn *et al.* report that by using digests to avoid storing redundant copies of payloads a Web cache can reduce its storage footprint by 15% and increase its hit rates [1]. Inktomi Corporation has patented such a scheme [18].

A variety of "duplicate suppression" schemes have been proposed for the Web. These differ from DTD chiefly in that 1) they are typically end-to-end mechanisms requiring the participation of orgin servers, whereas DTD can be used hop-by-hop at any level of a cache hierarchy, 2) they avoid the extra round trip that some variants of DTD suffer upon a miss, and 3) they can reduce but not eliminate redundant transfers. Mogul [19] reviews several duplicate suppression schemes (e.g., the Distribution and Replication Protocol (DRP) of van Hoff *et al.* [38]) and reported that they improve hit rates by modest margins, at best.

Previous studies have shown that redundant payload transfers on the Web are caused by complexities in the relationship between URLs and reply payloads (e.g., aliasing and rotation) [16], and by deficiencies in cache management algorithms and server-supplied metadata [41, 42].

Rhea *et al.* describe a sophisticated generalization of DTD called "Value-Based Web Caching" (VBWC) [28]. Whereas DTD operates on entire payloads, VBWC detects and eliminates redundant transfers at finer granularity by employing fingerprints calculated on variable-sized blocks. Block boundaries are computed as in Spring & Wetherall's approach [33]. In VBWC, editing a file affects only payload blocks in the immediate neighborhood of the change, ensuring that minor changes don't eliminate bandwidth savings. Rhea *et al.* implemented VBWC and evaluated it by polling seventeen popular Web sites; their evaluation also includes comparisons with delta encoding. They did not evaluate VBWC based on an actual client or proxy reference stream.

DTD sometimes entails an additional round trip between client and server, but requires no additional server state. By contrast, VBWC proxies must explicitly track client cache state in order to avoid the extra RTT except in rare circumstances. This is soft state, but it scales with both the number of clients and the size of responses, which makes VBWC less easily deployable than DTD. VBWC is also harder to evaluate using anonymized traces, because existing traces that include only MD5 digests of response bodies cannot be used to compute partial-payload fingerprints.

VBWC was designed to be run between an ISP's proxy and the end clients. While DTD can be used server-to-client or server-to-proxy, it can also be used proxy-to-client or proxy-to-proxy. In the latter cases, DTD imposes a store-and-forward cost (for computing the digest at the first proxy) on the entire payload, while VBWC's store-and-forward costs are per-block and thus potentially smaller. We do not yet know how significant these overheads are.

4 Duplicate Transfer Detection

Motivated by the wish to eliminate redundant HTTP transfers, we proposed "Duplicate Transfer Detection" (DTD). This solution applies equally to *all* redundant payload transfers, regardless of cause. Here we provide an overview of DTD (derived from [16]), and discuss several general design issues. In Section 5, we will present a more detailed protocol design, showing how DTD can be defined as a simple, compatible extension to HTTP/1.1 [9].

4.1 Overview of DTD

First, consider the behavior of a traditional HTTP cache, which we refer to as a "URL-indexed" cache, confronted with a request for URL U. If the cache finds that it does not currently hold an entry for that URL, this is a cache miss, and the cache issues or forwards a request for the URL towards the origin server, which would normally send a response containing payload P. (If the cache does hold an expired entry for the URL, it may send a "conditional" request, and if the server's view of the resource has not changed, it may return a "Not Modified" response without a payload.)

Now suppose that an idealized, infinite cache retains in storage every payload it has ever received, whether or not these payloads would be considered valid cache entries. A finite, URL-indexed cache differs from this idealization because it implements both an update policy (it only stores the most recent payload received for any given URL), and a replacement policy (it only stores a finite set of entries).

The concept behind Duplicate Transfer Detection is quite simple: If our idealized cache can determine, before receiving the server's response, whether it had ever previously received P, then we can avoid transferring that payload. Such a cache would suffer only compulsory misses and would never experience redundant transfers. A finite-cache realization of DTD would, of course, also suffer capacity misses.

How does the cache know whether it has received a payload P before the server sends the entire response? In DTD, the server (origin server or intermediate proxy cache) initially replies with a digest D of the payload, and the cache checks to see if any of its entries has a matching digest value. If so, the cache can signal the server not to send the payload (although the server must still send the HTTP message headers, which might be different). Thus, while DTD does not avoid the request and response message headers for a cache miss, it can avoid the transfer of any payload it has received previously. We say a "DTD hit" occurs when DTD prevents a payload transfer that would have occurred in a conventional URL-indexed cache.

An idealized DTD cache stores *all* payloads that it has received, and is able to look up a cached payload either by URL or by payload digest. In particular, it does not delete a payload P from storage simply because it has received a different payload P' for the same URL U. A realistic DTD cache, with finite capacity, may eventually delete payloads from its storage, based on some replacement policy.

4.2 What is a "payload"?

We have described DTD as operating on "payloads." In order to precisely specify DTD, we must also precisely specify the term "payload." That is, over what set of bytes is a digest calculated?

HTTP servers (the term "server" includes both origin servers and proxies) can send response messages containing either the full current value of a resource, a partial response containing one or more sub-ranges of the full value, or more complex partial responses (such as with delta encoding [21] or rsync [37]). HTTP responses can also be encoded using various compression formats, or with "chunked" encoding.

Whatever the format of the response, the ultimate client almost always wants to obtain a full current value of the referenced resource.[1] One of us introduced the term "instance" to mean "The entity that would be returned in a status-200 response to a GET request, at the current time, for [...] the specified resource," in an IETF standards-track document specifying how to extend HTTP/1.1 to support "instance digests" [20]. An instance consists of an "instance body" and some "instance headers."

Our DTD design equates "payloads" and "instance bodies." That is, servers provide instance digests, and a cache entry is indexed by the digest of the instance body it stores.

One could imagine an alternative in which DTD's digests are computed on HTTP message bodies, which might be partial responses. However, this seems less likely to eliminate redundant transfers; two partial responses for the same instance might not span the same range.

The "payloads are instance bodies" model works nicely with partial responses. For example, if a client requests bytes 0-10000 of URL X, and the server responds with a digest of the entire instance body, a DTD client checks its cache for a matching instance digest. If such an entry is found, the transfer can be avoided; the client can easily extract the required byte-range from its cache entry, rather than relying on the server's extraction.

Nothing in the DTD design prevents a cache from computing digests on non-instance data (such as partial responses, encoded responses, etc.) and matching incoming instance digests against cached non-instance data. Our intuition, however, is that such matches will occur too rarely to justify the additional overhead.

4.3 Deployment of DTD

DTD is best thought of as a hop-by-hop optimization of HTTP caching,[2] which can be implemented between any HTTP server and client (either one of which could be a proxy; DTD can be implemented between any data sender and receiver). In particular, DTD can be deployed unilaterally by an organization that controls both browser and proxy caches, e.g., AOL or MSN. It can also be deployed incrementally by any implementor of clients, servers, or proxies, because it is always optional for either end of a transfer. In the experiments described in Section 9 we demonstrate that DTD can be enabled purely through proxy modifications, if the origin server supports digest generation.

DTD's main requirement for server implementors is to compute and send instance digests. The algorithm used to compute the digest value D must not use too much server CPU time, and the digest representation must not consume too many bytes, or else the cost of speculatively sending digests will exceed the benefits of the DTD hits. Also, the digest must essentially never yield collisions, or else the client could end up with the wrong payload. A cryptographic hash algorithm such as MD5 [29] might have the right properties. We will assume the use of MD5 for this paper; Section 10.1 covers some issues in the choice of digest algorithm.

Note that DTD does not inherently require the client to compute any digests, if all servers send digests. However, to check against transmission errors or servers sending bogus digests, clients should probably compute digests anyway (see Section 10).

5 Protocol design issues

Our previous paper [16] briefly covered protocol design issues for DTD. In this section, we expand that discussion, including mechanisms for suppressing data transfer and specific HTTP mechanisms to support DTD.

5.1 Options for suppressing data transfer

One key aspect of DTD is the mechanism by which the client avoids receiving a payload, if the digest D matches an existing cache entry. This could be accomplished by deferring the transfer until the digest can be checked, or by aborting the transfer in progress if the digest matches some cache entry.

In the first category of approaches, the server sends the response headers but defers sending the payload until the client sends an explicit "proceed" message. In the other category, the server sends the payload immediately after the headers, but stops if the client sends an "abort" message. The "proceed" model imposes an extra round-trip time (RTT) on every cache miss, but never sends any redundant payload bytes. The "abort" model imposes no additional delays, but the abort message may fail to reach the server in time to save any bandwidth. Thus, the choice between alternatives requires consideration not only of implementation issues, but also of the magnitude of the RTT, and whether one is more concerned with optimizing bandwidth utilization or latency.

Each of these basic models allows several alternatives. These include:

Pure-proceed: Upon receiving the client's request, the HTTP server replies only with the HTTP headers (including digest D). The client sends a "proceed" message if D is not found in its cache, and the server sends the HTTP body (payload). Otherwise, no further messages are sent.

Proceed/don't bother: In the pure-proceed alternative, the server might need to buffer responses indefinitely, waiting for a possible "proceed" message. The "proceed/don't bother" alternative addresses this concern by allowing the client to send a "don't bother" message, if digest D *does* match a cache entry; the message allows the server to free the buffer more quickly.

Auto-proceed for short responses: The proceed model risks exchanging an extra set of headers and delaying an extra RTT. For short payloads, the transfer time saved by a DTD hit might not be worth this overhead. The server could optimize the short-payload case by sending the payload immediately for payload sizes below a threshold.

Abort: The server sends the payload immediately after the HTTP headers (as in normal HTTP operation). The client sends a special HTTP "abort" message if digest D matches a cache entry, telling the server to terminate the transfer as soon as possible.

Note that in the proceed model, not every payload need be delayed. Web pages often include multiple images; for example, we previously found 8.5 image references per HTML reference in an uncached reference stream, and 1.9 images per HTML reference in a client-cached stream [16]. A client that pipelines [26] its requests for images can also pipeline its "proceed" messages. Thus, the extra RTT delay can be amortized over all of the images on a Web page, rather than being paid once per image.

In this paper, we examine only the pure-proceed model, for reasons of space and simplicity.

5.2 Extending HTTP to support DTD

The changes required to extend HTTP/1.1 [9] to support DTD depend on which transfer-suppression approach is chosen. The "pure proceed" approach to DTD can be implemented without any changes to HTTP/1.1 beyond existing IETF standards-track proposals.

The client first uses mechanisms specified in the Proposed Standard for instance digests [20] to obtain current instance headers, including an instance digest. It obtains these via a HEAD request, which prevents the server from sending an instance body [9, Section 9.4]. If the client finds no cache entry with a matching instance digest, or if a non-DTD server fails to return a digest, the client simply issues a GET request to obtain the full instance body.

This protocol design, while simple, has several drawbacks:

- **It potentially adds one extra RTT per miss:** The client sends both a HEAD and a GET request on a DTD miss, so this could add an extra RTT of latency per request. In practice, most HTTP requests are for images embedded in HTML pages, which allows an HTTP/1.1 client to pipeline some or all of a page's image requests in one transmission (and the server can likewise batch the HEAD and GET responses). So for typical compound Web pages the pure-proceed approach adds *at most* two additional mandatory RTTs: one for the HTML container and one more for *all* of the embedded images.
- **It adds an extra set of request and response headers per miss:** This cuts into the bandwidth savings offered by DTD. Therefore, DTD is not worth doing if the mean savings (in response-body bytes) is smaller than the sum of the mean request and response header lengths (see Section 6.1).
- **It depends on request idempotency:** If the (HEAD, GET) sequence had different side effects than a single GET request on the same URI, this would give DTD incorrect semantics. The HTTP/1.1 specification recommends that "the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval," [9, Section 9.1.1], but some sites might ignore this recommendation. If so, DTD clients might need to apply some heuristics, such as not issuing the extra HEAD request on URLs containing "?", or (perhaps) using DTD *only* for embedded images.
- **The server might never send a digest:** HTTP servers are not required to send instance digests, and there is no (current) mechanism to discover if a server would ever send one. The client could thus incur all of the costs listed above, with respect to a given server, without ever gaining a benefit. Clients might need to cease using this DTD approach with any server that fails to send a digest after some threshold number of requests.

Figure 1 shows an example of the HTTP messages between a client and server for a DTD miss. For a DTD hit, the second pair of messages would simply be omitted. The `Want-Digest` and `Digest` headers are described in RFC 3230 [20]; all other headers are standard in HTTP/1.1 [9].

Using `Want-Digest` and `Digest` is the "right" implementation of DTD, because it works even for partial-content responses, is extensible to digest algorithms other than MD5, and avoids unnecessary digest computations at the origin server. But since RFC 3230 is not widely implemented, we tested DTD using the `Content-MD5` support available in major Web servers (e.g., Apache and IIS). This is sub-optimal because it does not allow the server to avoid computing MD5s when the client has no use for them.

The pure-proceed approach is equally usable hop-by-hop or end-to-end, because any intermediate proxy can generate or check digests. (A proxy-to-proxy implementation must use `Digest` because HTTP/1.1 [9, section 14.15] specifically prohibits proxies from adding `Content-MD5`.) Note that proxy-to-client or proxy-to-proxy DTD could impose an extra store-and-forward delay, while the first proxy computes the digest header. (Some existing proxies might already buffer short responses, in any case.)

6 Trace-based performance analysis

Section 9 presents measured performance of an actual DTD implementation. However, those measurements are driven from a synthetic reference stream, which cannot prove how frequent redundant transfers are in real-world workloads. Here we analyze two real-world traces to show how many redundant transfers, and how many bytes, could be eliminated by DTD.

Relatively few existing client and proxy HTTP traces include the response body digests we needed for our analysis. For example, the trace used by Douglis *et al.* [5] may have been lost in a disk crash; other such traces are unavailable due to proprietary considerations. We re-analyzed the anonymized client and proxy traces from our prior study [16]. These were collected, respectively, at WebTV Networks in September 2000 and at Compaq Corporation in early 1999. The WebTV trace was made with client caches disabled; both traces were made without proxy caching. Both traces include an MD5 digest for each payload transferred. The WebTV trace in-

First client request:
```
HEAD /images/logo.gif HTTP/1.1
Host: example.com
Want-Digest: MD5
```

First server response:
```
HTTP/1.1 200 OK
Date: Tue, 30 Jul 2002 18:30:05 GMT
Digest: md5=HUXZLQLMuI/KZ5KDcJPcOA==
Cache-control: max-age=3600
ETag: "xyzzy"
```

Second client request:
```
GET /images/logo.gif HTTP/1.1
Host: example.com
```

Second server response:
```
HTTP/1.1 200 OK
Date: Tue, 30 Jul 2002 18:30:06 GMT
Digest: md5=HUXZLQLMuI/KZ5KDcJPcOA==
Cache-control: max-age=3600
ETag: "xyzzy"
```

(message body omitted)

Figure 1: Example of HTTP messages (pure-proceed approach).

cludes 326 million references from 37 thousand clients to 33 million URLs on 253 thousand servers over sixteen days; the Compaq trace includes 79 million references from 22 thousand clients to 20 million URLs on 454 thousand servers over 90 days. Many further details of these traces are described in [16] and are omitted here for space reasons.

Given a request for URL X that results in reply instance body B, the following properties may or may not hold:

i) there exists some URL Y such that $Y \neq X$ and B was the most-recent instance body for Y.
ii) there exists some URL Z such that $Z \neq X$ and B was a past instance body for Z, but not the most recent.
iii) B was a past instance body for X, but not the most recent.
iv) B was the most recent instance body for X.

Properties (iii) and (iv) are mutually exclusive, but any other combination is possible, so a total of twelve possibilities exist: a given transaction might have none of these properties (if it has never been seen before), or several at once (e.g., both most recent for X and most recent for $Y \neq X$).

We analyzed both the WebTV and Compaq traces according to this categorization. The results are in Tables 1 and 2 respectively. The cold-start results cover the entire traces. Consistent with our earlier methodology [16], for the warm-start results we (only somewhat arbitrarily) warm the simulated cache with the first 186 million references (for WebTV) or 50 million references (for Compaq).

In the WebTV warm-start results, 10% of the transfers involve payloads never before seen in the trace ("new payloads"); these will miss in any kind of cache. Another 87% have property (iv), for which a traditional, infinite cache with perfect revalidation would avoid a payload transfer. (This "hit rate" seems high, but remember that the WebTV trace was made with client caches disabled.) The remainder, about 3%, are transfers that DTD would avoid. In other words, a traditional URL-indexed cache would see a miss rate of at least 13%, compared to a DTD-cache miss rate of 10%; DTD would eliminate 23% of a conventional cache's misses.

In the Compaq warm-start results, 37% are new payloads, and 55% have property (iv). The remainder, about 8%, are transfers that DTD would avoid. A traditional cache would see a miss rate of 45%, versus a DTD-cache miss rate of 37%; DTD would eliminate roughly 18% of a conventional cache's misses for this trace.

If we restrict the DTD implementation to save at most one entry per URL (i.e., to store no more entries than a traditional cache), then the DTD cache will require transfers for properties (ii) and (iii), but will still avoid transfers for property (i). In this situation, DTD would avoid 2.6% of the transfers in the WebTV trace, and 5.8% of the transfers in the Compaq trace, assuming a warm cache. (These values are the sums of the *Warm-start Transfers* column for rows where property (i) holds and property (iv) does not.)

Weighting the results by bytes transferred better describes bandwidth savings, of course. Looking just at the warm-cache data, new (i.e., mandatory-transfer) payloads account for 30% of the WebTV bytes, and 57% of the Compaq bytes. Variations of property (iv), hits for a perfect traditional cache, account for 64% of the WebTV bytes, and 34% of the Compaq bytes. The transfers that DTD would avoid account for 5% of the WebTV bytes, and 9% of the Compaq bytes.

In other words, a traditional URL-indexed cache would see a byte-weighted miss rate of at least 36% for the WebTV trace, compared to a DTD-cache miss rate of 30% (66% vs. 57% for the Compaq trace). In terms of the *reduction* in the number of bytes sent from the origin server, DTD would save (relative to a URL-indexed cache) 15% for the WebTV trace, and 14% for the Compaq trace.

6.1 Overheads from the proceed model

Because the proceed model for DTD causes an extra pair of request and response headers when the digest does not match, to evaluate the overall byte-transfer savings for this model we must compare the bytes saved by DTD (for properties (ii) and (iii)) with the number of

property				Cold-start		Cold-start		Warm-start		Warm-start		
iv	iii	ii	i	Transfers	%	MBytes	%	Transfers	%	MBytes	%	Current reply payload was...
0	0	0	0	36,573,310	11.22	609,935	32.40	13,915,207	9.94	245,010	30.40	never returned before
0	0	0	1	6,047,586	1.85	39,205	2.08	2,332,816	1.67	15,735	1.95	most-recent for other URL
0	0	1	0	94,375	0.03	1,937	0.10	43,313	0.03	1,066	0.13	returned for other URL, not most recent
0	0	1	1	2,070,537	0.64	8,820	0.47	908,075	0.65	3,715	0.46	
0	1	0	0	1,048,493	0.32	35,074	1.86	465,865	0.33	16,906	2.10	returned for current URL, not most recent
0	1	0	1	129,349	0.04	3,089	0.16	62,776	0.04	1,551	0.19	
0	1	1	0	150,533	0.05	2,189	0.12	67,477	0.05	1,093	0.14	
0	1	1	1	681,840	0.21	3,350	0.18	309,030	0.22	1,655	0.21	
1	0	0	0	131,262,060	40.26	662,120	35.17	52,607,080	37.56	272,289	33.79	most recent for current URL
1	0	0	1	138,927,549	42.61	490,892	26.08	64,263,811	45.88	231,911	28.78	
1	0	1	0	290,628	0.09	2,202	0.12	168,472	0.12	1,143	0.14	
1	0	1	1	8,784,417	2.69	23,740	1.26	4,916,756	3.51	13,857	1.72	
				326,060,677		1,882,552		140,060,678		805,928		Totals

Table 1: WebTV trace categorization.

property				Cold-start		Cold-start		Warm-start		Warm-start		
iv	iii	ii	i	Transfers	%	MBytes	%	Transfers	%	MBytes	%	Current reply payload was...
0	0	0	0	30,591,044	38.77	512,562	59.53	10,575,651	36.58	182,372	56.56	never returned before
0	0	0	1	3,504,391	4.44	49,967	5.80	1,291,369	4.47	21,793	6.76	most-recent for other URL
0	0	1	0	148,533	0.19	1,357	0.16	49,339	0.17	490	0.15	returned for other URL, not most recent
0	0	1	1	604,076	0.77	2,721	0.32	229,799	0.79	1,146	0.36	
0	1	0	0	1,554,331	1.97	10,521	1.22	612,795	2.12	3,741	1.16	returned for current URL, not most recent
0	1	0	1	130,356	0.17	1,010	0.12	48,965	0.17	430	0.13	
0	1	1	0	164,992	0.21	1,091	0.13	62,984	0.22	432	0.13	
0	1	1	1	264,100	0.33	1,812	0.21	112,359	0.39	789	0.24	
1	0	0	0	20,492,740	25.97	166,360	19.32	7,230,114	25.01	59,824	18.55	most recent for current URL
1	0	0	1	19,126,071	24.24	106,183	12.33	7,587,555	26.24	47,811	14.83	
1	0	1	0	165,425	0.21	943	0.11	65,695	0.23	397	0.12	
1	0	1	1	2,167,290	2.75	6,442	0.75	1,046,725	3.62	3,198	0.99	
				78,913,349		860,970		28,913,350		322,421		Totals

Table 2: Compaq trace categorization.

extra header bytes spent on the new-payload transfers. We can ignore property (iv) by assuming that these references could be cache hits. DTD (warm-start) saves a mean of 3036 bytes of payload transfer for each new-payload reference in the WebTV trace (warm-start), and 2857 bytes for each new-payload reference in the Compaq trace. These savings are much larger than the mean request+response header sizes reported in previous studies (e.g., [6, 13]) [3] so the proceed model does not waste too much of the potential savings.

DTD requires digests in response headers (for MD5, 24 bytes plus about 10 bytes of syntax overhead), which further reduces savings. However, digests are useful for integrity checks, and so might be sent even without DTD.

6.2 If-None-Match with multiple entity tags

HTTP/1.1 supports the use of *entity tags* to validate cache entries: a server may provide an instance-specific entity tag in the `ETag` response header, and a client may send this entity tag back to the server in an `If-None-Match` request header to check if its cache entry is still valid. `If-None-Match` may carry multiple entity tags, in which case the server can return "304 Not Modified" (along with the current entity tag) if any of those tags is current.

This feature would allow a non-DTD cache to avoid transfers when property (iii) holds. Referring to the warm-start columns in Tables 1 and 2, we see that this could avoid at most 0.6% of the transfers and 2.6% of the bytes for the WebTV trace, and 2.9% of the transfers and 1.7% of the bytes for the Compaq trace.

However, these are upper bounds, since this simple analysis assumes that every response carries an entity tag, and the servers always use exactly one entity tag per distinct instance body. Neither is true in practice; only 66% of the responses in the WebTV trace carried entity tags, and we know that some servers can assign different entity tags to identical instance bodies. In summary, DTD avoids transferring significantly more bytes than could be avoided using multiple entity tags in `If-None-Match`.

6.3 Multiple cache entries per URL?

The full benefit of DTD accrues when the cache stores more than one payload per URL. The most natural clean-slate DTD cache design treats *payloads* rather than URLs as the basic storage type. URLs are merely one way to index into this underlying store; payload digests are another. The cache may therefore store multiple payloads for a given URL, and also payloads that are not the most-recent response for *any* URL (as in the case of rotated resources). These properties, while desirable, might be difficult to retrofit onto some legacy cache implementations; how much do they help? It helps for references that have either property (ii) or (iii) while having neither property (i) nor (iv). These represent just 0.4% of the warm-start transfers in the WebTV trace, but 2.5% of the warm-start transfers in the Compaq trace, so it probably is useful to store multiple payloads per URL.

7 Model-based latency analysis

The analysis in Section 6 concentrates on the number of bytes that could be saved using DTD, which may be of economic interest to network operators. End users, however, care more about latency. Predicting the latency effects of change to Web protocols can be difficult, since so many variables can affect overall latency.

We have developed a simple model for understanding when pure-proceed DTD might improve latency over a traditional Web cache. This model ignores issues such as response pipelining, network congestion, TCP algorithms such as slow-start, and correlations of the hit ratio and duplication ratio with other parameters, but it can help guide intuition.

Given these parameters:

$$\begin{aligned} RTT &= \text{round trip time, cache to server} \\ BW &= \text{effective link bandwidth, bits/sec} \\ L_{\text{resp}} &= \text{response length, bits} \\ HR_{\text{Conv}} &= \text{conventional-cache hit ratio} \\ HR_{\text{DTDonly}} &= \text{DTD-only hit ratio} \\ T_{\text{lookup}} &= \text{Cache-lookup latency} \end{aligned}$$

then we can derive these latencies (if we over-simplify by assuming that HTTP headers are negligible in length):

$$\begin{aligned} T_{\text{ConvHit}} &= T_{\text{lookup}} \\ T_{\text{ConvMiss}} &= T_{\text{lookup}} + RTT + L_{\text{resp}}/BW \\ T_{\text{DTDonlyHit}} &= T_{\text{lookup}} + RTT + T_{\text{ConvHit}} \\ T_{\text{DTDMiss}} &= T_{\text{lookup}} + RTT + T_{\text{ConvMiss}} \end{aligned}$$

The extra RTT in $T_{\text{DTDonlyHit}}$ and T_{DTDMiss} comes from the HEAD operation that a DTD cache performs after the conventional lookup misses. The extra T_{lookup} in $T_{\text{DTDonlyHit}}$ and T_{DTDMiss} comes from the need to do lookups both on the URL and the digest in those cases.

We simplify by assuming that $T_{\text{lookup}} = 0$, a reasonable approximation for a well-implemented cache.

We can then express the expected latencies for conventional and DTD caches:

$$\begin{aligned} E_{\text{Conv}} &= HR_{\text{Conv}} \times T_{\text{ConvHit}} \\ &\quad + (1 - HR_{\text{Conv}})T_{\text{ConvMiss}} \end{aligned}$$

Scenario	RTT	Bandwidth	Break-even response size (bytes) WebTV	Compaq
Cellphone	100ms	10Kb/s	415	549
Modem	100ms	56Kb/s	2325	3075
DSL	30ms	384Kb/s	4783	6325
WAN	42ms	6000Kb/s	104629	138367

Table 3: Examples of model output.

$$\begin{aligned} E_{\text{DTD}} &= HR_{\text{Conv}} \times T_{\text{ConvHit}} \\ &\quad + HR_{\text{DTDonly}} \times T_{\text{DTDonlyHit}} \\ &\quad + (1 - (HR_{\text{Conv}} + HR_{\text{DTDonly}}))T_{\text{DTDMiss}} \end{aligned}$$

DTD improves the expected latency if $E_{\text{DTD}} < E_{\text{Conv}}$, which (by algebra) is true if

$$BW < \frac{L_{\text{resp}} \times HR_{\text{DTDonly}}}{RTT(1 - (HR_{\text{Conv}} + HR_{\text{DTDonly}}))} \qquad (1)$$

DTD is thus more likely to pay off as the effective link bandwidth and/or RTT decrease, and as the transfer length and hit ratios increase.

We evaluated Equation 1 using warm-cache hit-ratio values taken from the WebTV and Compaq trace analyses in Tables 1 and 2 and various combinations of RTT and bandwidth. Table 3 shows the results for several scenarios: "cellphone," "modem," "DSL," and "WAN," corresponding respectively to the results shown later in Figures 4(a), 4(b), 4(c), and 6. The break-even response sizes shown in the table imply that DTD would improve latency on cellphone and modem links, and perhaps on DSL links, given the typical mean response sizes summarized in Table 4 of [16]. DTD would hurt latency on high-speed WAN links except if its use were restricted to relatively large responses.

8 Implementation design and experience

Most of the new code required for DTD, using the proceed model, is located in cache implementations. (We also needed server support for digests; we relied on existing support for `Content-MD5`, which is only partially appropriate; see Section 5.2.) Both clients (browsers) and proxies have caches; for our experiments, we limited ourselves modifying a proxy cache server. By running a "private" proxy cache co-located with a browser, we can emulate most of the benefits of integrating DTD into a browser cache. (It should be simpler to add DTD to a browser cache than it was to add it to a proxy cache.)

We chose to implement the pure-proceed approach to DTD as modifications to the Squid proxy server [34] (version 2.4.STABLE7). Our code is available from `http://devel.squid-cache.org/dtd/`. The major changes we made are:

- Creating a "payload" datatype separate from a cache entry. This inverts the existing data-structure dependence between a payload and a URL.

- Indexing into the payload database by digest as well as by URL.
- Generating a preliminary HEAD request to obtain the server's digest.
- Checking the returned digest for DTD-related HEAD requests, and generating a GET request if the digest is not found in the cache, or if no digest is returned.

Our modified Squid uses "Duplicate Storage Avoidance" (DSA). Each distinct payload (i.e., with a given digest) is stored only once; if the payload is current for several URLs, the URL-indexed entries incorporate the payload by reference (see `http://devel.squid-cache.org/dsa/`).

The DTD and DSA changes together involve about 3420 lines of mostly simple but tedious "diffs" to Squid; much of the new code represents modified versions of existing Squid code. About one third of the new lines are pre-processor directives (e.g., "#ifdef").

A cache that supports partial content (HTTP status-206 responses) must be careful not to associate an entire-instance digest with a stored partial-instance body, or else DTD could unwittingly supply incomplete bodies. Our implementation does not yet support partial content.

In hindsight, the choice to modify Squid may have been a mistake. The existing Squid code is extremely complex and hard to understand, and we found many bugs in our own code that resulted from our failure to maintain poorly documented invariants expected by the rest of Squid. We know some bugs remain.

9 Experimental results

The analysis in Section 6, based on traces of real users, predicts the bandwidth savings from DTD, but cannot tell us how DTD affects latency. To help answer this question, we ran experiments using our modified version of Squid.

9.1 Experimental design

We tested our DTD implementation in two different environments The first was an "Emulated-WAN" environment, in which the two systems (server, proxy+client) were physically close, and connected by a 10 Mbit/sec LAN. We then emulated a variety of WAN environments using the Dummynet [30] feature of FreeBSD, which allowed us to choose a variety of latency and bandwidths between the server and proxy, enabling us to measure how DTD performance varies with network characteristics. The second was a "Real-WAN" environment, using a server at Worcester Polytechnic Institute (WPI) in Massachusetts, while the DTD-capable proxy and the client ran on a system at the University of Michigan.

In our tests, we ran the proxy (modified or unmodified) on the same system as the client, to simulate the use of a client cache with or without support for DTD. All systems were otherwise unloaded, except for the real-WAN origin server.

All of the hosts ran Linux, except for the emulated-WAN server which ran FreeBSD. The server at WPI uses Apache/1.3.12, while the emulated-WAN server uses Apache/2.0.47. For the emulated-WAN experiments, the proxy/client was a 550 MHz Pentium III and the server was a 466 MHz AlphaServer DS10L. For the real-WAN experiments, the proxy/client was a 4-CPU 450 MHz Pentium II and the server was a 600 MHz Pentium III.

We measured a mean RTT of 42 msec for the real-WAN path, and approximate effective bandwidths from 6.1 to 7.7 Mbits/sec. In the emulated-WAN tests, we used Dummynet to impose symmetric RTTs of 0, 30, and 100 msec, and bandwidth limits of 10K, 56K, 384K, 1.5M, and 10M bits/sec.

We ran trials with file (body) sizes (not including HTTP headers) of 2^i bytes, for $i = 10, 11, \ldots, 20$; i.e., between 1KB and 1MB.[4] Each file byte was derived from a pseudo-random number generator, thus making it difficult for any network element (such as a modem) to compress the files and change their effective transfer sizes.

For each combination of network characteristics and body size, we ran experiments using three different proxy configurations: no proxy, unmodified Squid, and our DTD-capable modified Squid proxy. With unmodified Squid, we ran trials where the references were arranged to be compulsory cache misses, and trials where the references were guaranteed to be cache hits. With our DTD-capable Squid, we ran compulsory-miss, guaranteed-hit, and DTD-only-hit trials; the last category were references where we arranged that the cache contained an entry with a matching digest value, but not a matching URL. We arranged compulsory cache misses by restarting the proxy software with a cold cache as necessary; we arranged guaranteed hits by careful choice of the reference sequence, and by ensuring that the working set was much smaller than the cache size.

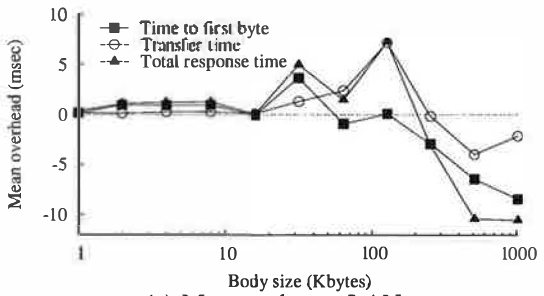
In each set of experiments, we measured end-to-end response time using httperf [22]. This program reports the latency between issuing a request and receiving the first byte of the response headers (time-to-first-byte, or TTFB), as well as the latency between receiving the first byte of the response headers and the last byte of the response body (transfer duration, or TD). For the 1KB body size, the headers and body might fit into one packet, in which case TD would be negligible. The total response latency is thus TTFB+TD. In each trial, we used httperf to fetch "bunches" of 10 distinct files with the same length.

For a given network configuration, we measure latencies for one bunch for each combination of body size and proxy configuration, then repeat that set of measurements

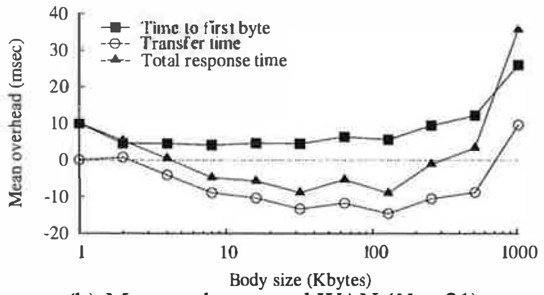
N times. Results in this paper show the mean for $N = 9$ unless otherwise noted.

9.2 Measured overheads

The use of a proxy server introduces overheads that would not be present if our DTD implementation were integrated into a client cache. Also, Squid is known to add significant latency due to fundamental design choices [17]. We can estimate the overheads imposed by our implementation strategy of using Squid rather than an integrated client cache; we do this by comparing the no-proxy latencies with the latencies for cache-miss retrievals via unmodified Squid.



(a) Measured over LAN



(b) Measured over real WAN ($N = 21$)

Figure 2: Overhead imposed by unmodified Squid

Figure 2 shows the overheads imposed by unmodified Squid connected to the server over both a full-speed LAN and over the WAN path described above. In the LAN case, Squid adds almost no latency larger than the trial-to-trial measurement errors (which cause some of the negative "overheads" in Figure 2(a); these errors are below 2% of the total latencies). Overheads from our WAN tests (Figure 2(b)) are harder to interpret, although using unmodified Squid seems to consistently *improve* the transfer times for most body sizes. This effect also holds when we run experiments using an emulated WAN with similar delay and bandwidth. We cannot offer a plausible explanation, but because most of the results in this paper compare performance for our modified Squid against the unmodified version, rather than against the no-proxy case, we leave this mystery to others.

On our LAN, the TTFB latency difference between a Squid miss and a no-proxy operation, for most body sizes, is about 1 msec. This places an upper bound on the cache-lookup latency, because Squid imposes other overheads beyond this lookup, and so confirms our assumption in Section 7 that the lookup latency is negligible.

DTD requires the origin server to send the digest of the payload (body). In our experiments, we use the MD5 digest algorithm, whose computation imposes some cost [36]. In principle, servers could cache MD5 computations for frequently-accessed content. Also, Moore's Law suggests that MD5 computation will decline in cost relative to speed-of-light latencies. However, current servers (such as Apache) do not cache MD5 values, so the use of DTD adds this computational overhead. We quantified the cost by comparing the latencies for no-proxy retrievals with and without MD5 computation enabled at the Apache server.

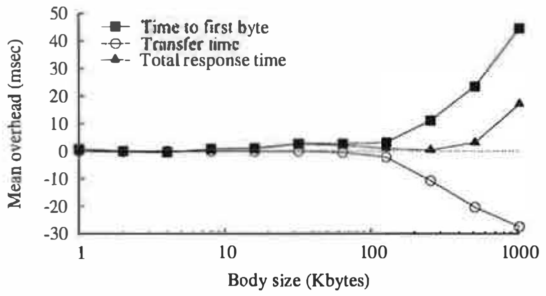


Figure 3: Overhead for MD5 computation

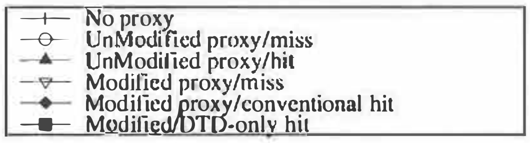
Figure 3 shows the overheads that MD5 imposes for a LAN-based, proxyless configuration. For bodies smaller than 128 KBytes, the overheads are negligible (under 3 msec). For larger bodies, MD5 computation adds measurable overhead, but still less than a tenth of the absolute response time (e.g., 1218 msec for 1024-byte bodies). The increase in response time is *smaller* than the increase in TTFB for these larger sizes, probably because the MD5 pass effectively prefetches the file into the server's file buffer; this prefetching (as Figure 3 implies) makes the TCP transfer slightly more efficient.

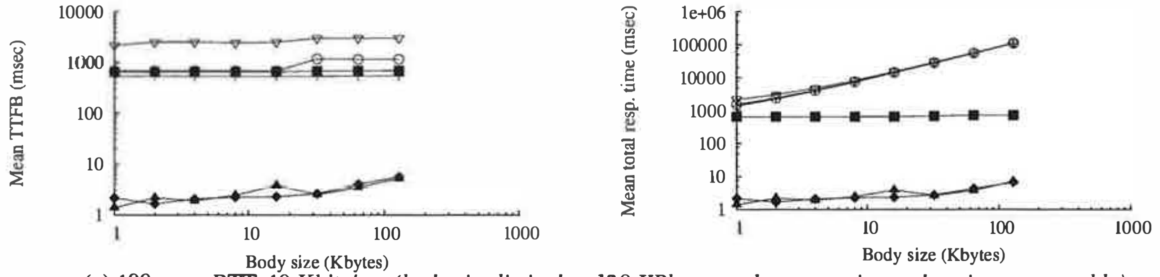
9.3 Emulated-WAN experiments

Figure 4 shows time-to-first-byte and total response time results, in the left and right columns respectively, for selected emulated-WAN experiments. For reasons of space, we only show results for: ($RTT = 100msec$, $10Kbits/sec$), a plausible cell-phone link; ($RTT = 100msec$, $56Kbits/sec$), a typical dialup modem; ($RTT = 30msec$, $384Kbits/sec$), a DSL connection to a regional server; and ($RTT = 100msec$, $10Mbits/sec$), a bad case for DTD because the RTT and bandwidth are both high.

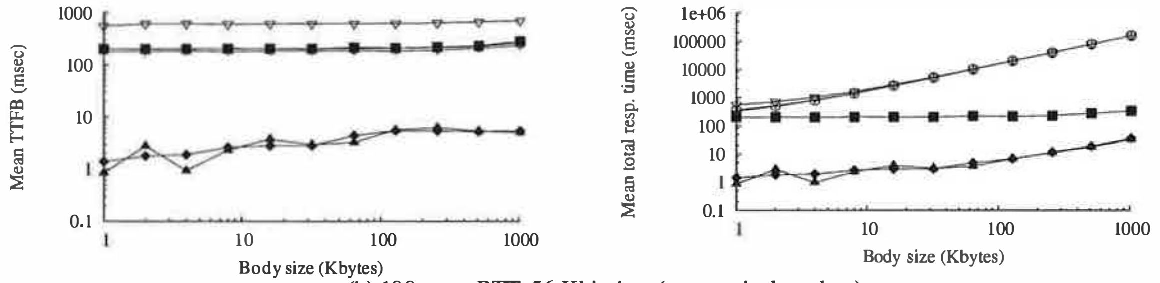
For all combinations of network parameters that we tested, the TTFB latency for a DTD-only hit is slightly above one RTT (approximately the TTFB of a cache miss), as we would expect from the cost of the HEAD

■ Key for all graphs in this figure

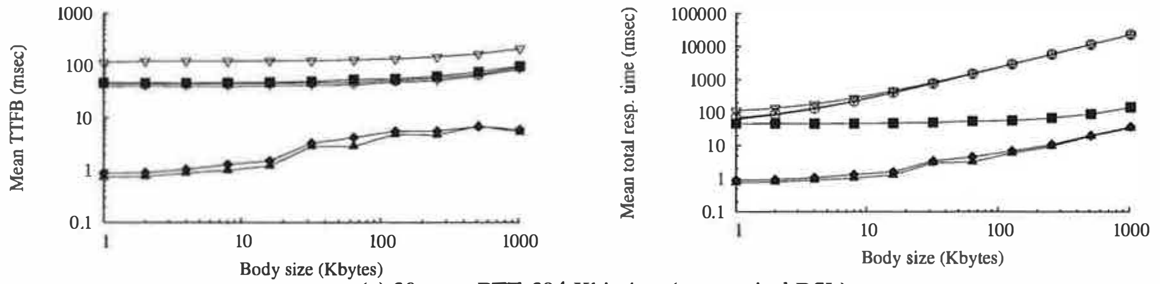




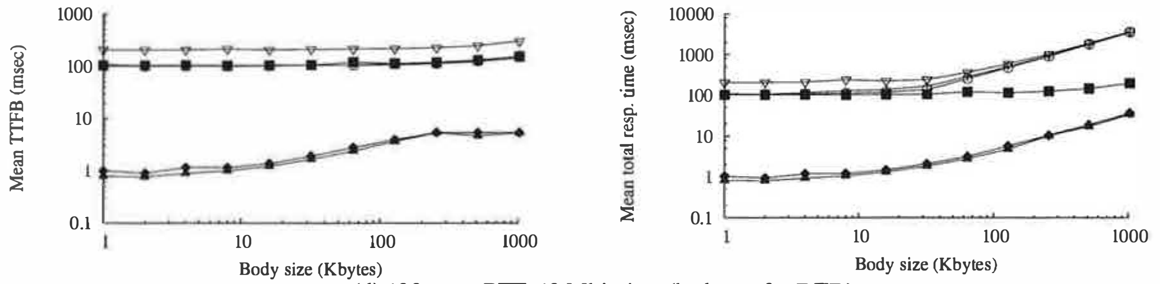
(a) 100 msec RTT, 10 Kbits/sec *(body size limited to 128 KBbytes, to keep experiment durations reasonable)*



(b) 100 msec RTT, 56 Kbits/sec (e.g., typical modem)



(c) 30 msec RTT, 384 Kbits/sec (e.g., typical DSL)



(d) 100 msec RTT, 10 Mbits/sec (bad case for DTD)

Figure 4: Emulated-WAN results

operation. The total response latency for a DTD-only hit is also approximately one RTT, because no body is transferred from origin server to cache. (The cache is co-located with the client, so there is almost no transfer cost between those agents.)

The total latency for compulsory miss by a DTD-capable cache will be one RTT higher than that of a traditional cache. This is clearly visible in the left column of figures (the log scale makes it less visible in the right column, where results are sometimes dominated by bandwidth-induced delays). This is a penalty that a DTD cache must make up by its improved latency on DTD-only hits, with respect to the conventional misses that they displace.

A DTD-only hit should never have a higher total latency than a conventional miss by a non-DTD cache, but it can be much lower if the conventional miss incurs a large transfer cost. For example, in Figure 4(a–c), at a body size of just 8 KBytes, the total latency is significantly lower for a DTD-only hit than for a conventional miss. In Figure 4(d), however, DTD shows no latency benefit except for very large body sizes, because the high bandwidth minimizes transfer cost, while the high RTT dominates total latency.

Note that while Figure 4 shows that DTD-only hits can be much faster than the conventional misses they replace, without knowing the various hit ratios (see Section 7) one cannot infer whether DTD provides a net benefit.

9.4 Real-WAN experiments

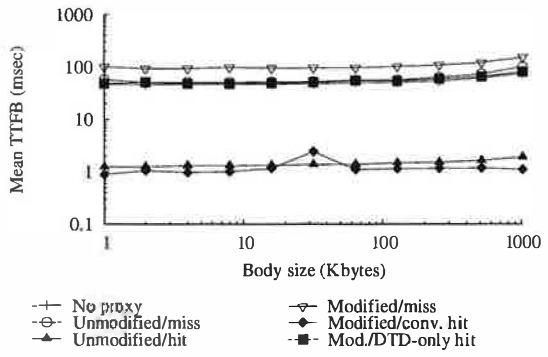


Figure 5: Real WAN – Time-to-first-byte results

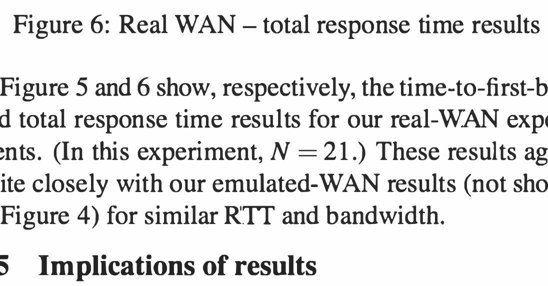


Figure 6: Real WAN – total response time results

Figure 5 and 6 show, respectively, the time-to-first-byte and total response time results for our real-WAN experiments. (In this experiment, $N = 21$.) These results agree quite closely with our emulated-WAN results (not shown in Figure 4) for similar RTT and bandwidth.

9.5 Implications of results

Our experimental results generally confirm the analytic model in Section 7, although our experiments do not attempt to model miss ratios.

We can evaluate whether DTD is beneficial at a particular point in the parameter space. For this example, we assume the miss ratios reported from the WebTV trace in Section 6, 10% for a DTD client cache vs. 13% for a conventional client cache, and assume that these ratios are independent of response size. Using the results in Figure 4(b), a modem user with ($RTT = 100msec$, $56Kbits/sec$) who retrieves a number of 8 KByte files would have a mean expected latency improvement using DTD of about 15 msec (compared to an overall expected mean, without DTD, of 185 msec). The same user retrieving a number of 32 KByte files would see a mean improvement of 126 msec (vs. an overall non-DTD mean of 661 msec).

A user on a slower network, with ($RTT = 100msec$, $10Kbits/sec$), would see even larger improvements from DTD. However, a user of our relatively good WAN connection would see a net latency loss from DTD for body sizes below a break-even point of about 64 KBytes. Since most Web responses are smaller than that, on good WAN links one might only want to use DTD for special tasks such as downloading software (the original motivation for DRP [38]).

10 Security considerations

Measures that improve the performance of computing systems often create subtle security vulnerabilities, and caching is a prime example. Timing attacks on processor memory hierarchies have been known for decades, e.g., the famous TENEX password attack [35, pp. 183–4]. Recently Felten *et al.* have described variants applicable to Web caching [8]. DTD adds at least two additional security problems.

First, if an attacker can generate payload digest collisions, then she can cause a DTD proxy to deliver incorrect payloads. The details are omitted here but are available in [14]. The attack is straightforward and can be prevented through the use of secure message digest functions (see Section 10.1).

A more subtle problem involves information leakage; interestingly, the attack does *not* rely on timing information of any kind.[5] A server can exploit DTD to learn the contents of a client's cache:

1. User Bob's browser and the `nosy.com` server employ DTD.
2. Bob issues a request for uninteresting URL `http://nosy.com/humdrum.html`.
3. `nosy.com` replies with digest(`naughty.gif`), even though it never receives or serves requests for this interesting payload.
4. Bob's browser fails to retrieve the full payload, thereby revealing that Bob already has it.

Sophisticated implementations of this attack might employ JavaScript within HTML pages to systematically search a client's cache for interesting payloads, analogous to the timing attacks described by Felten *et al.* [8]. Attacks of this form can be *detected* easily, by simply retrieving a full payload and verifying the digest previously obtained from the server. Furthermore such attacks can be *avoided* if the client simply refrains from employing DTD when communicating with untrusted sites. Another possible countermeasure is to employ DTD only *within* sites; in the example above, Bob's browser would always fetch payloads except when it found a match supplied by the same server. This ensures that DTD reveals nothing about Bob's surfing that the server doesn't already know. However this approach may severely limit the benefits of DTD, because most aliasing occurs *across* sites rather than within sites [16].

10.1 Choice of digest algorithm

DTD would be unreliable if the digest function were prone to accidental collisions under normal usage. MD5 might not be sufficient for widespread deployment; if not, one could achieve an arbitrarily low rate of accidental collisions by increasing the hash size, at the cost of slightly higher overheads. (Henson [12] discusses some risks associated with digest-based protocols; we disagree with some of the conclusions in that paper.)

DTD would be vulnerable to attack if it were computationally feasible to generate digest collisions deliberately. Our work has assumed the use of MD5 [29], but MD5's collision-resistance has been questioned [31]. Other algorithms, such as SHA1 [24], might be more appropriate.

11 Future work

We see many possible extensions of this work. We would like to explore and evaluate the protocol alternatives in Section 5, and perhaps to unify DTD with similar techniques such as rsync [37]. We would also like to see the trace-based analysis of Section 6 applied to a broader set of traces. One could also improve on our synthetic benchmarks by using miss-ratio and response-length distributions taken from traces.

Neither our model nor the original Squid code base supports pipelining, which is known to benefit HTTP performance in general [25], and ought to improve the tradeoff in favor of DTD; evaluation of a pipelined DTD cache would require shifting to a new code base.

Because a DTD cache, unlike a traditional cache, might store multiple entries per URL, cache replacement policies designed for traditional caches might interact poorly with DTD. We suspect that the most natural replacement policy for DTD is to redefine an existing policy with respect to unique instances rather than to URLs. While we have not yet evaluated such policies, we believe that a DTD cache with such a policy will not suffer a higher miss rate than a conventional URL-indexed cache with the analogous policy.

12 Summary and conclusions

This paper has described how Duplicate Transfer Detection can be implemented in HTTP without explicit protocol changes, and briefly sketched several alternative designs. We showed, using two real-world traces, how DTD could reduce miss rates and bandwidth requirements—14% to 15% of the bytes transferred in our traces. We provided a simple model to show when use of DTD should reduce expected latency relative to a conventional cache. We described a simple implementation of DTD for Squid. Using tests of real and emulated WANs, we showed measurements that clarify the conditions under which DTD reduces overall latency. For realistic hit ratios and response sizes, DTD does provide a net latency benefit for some common network environments.

Acknowledgments

We thank Microsoft for allowing us to study their data, especially Stuart Ozer, Arnold de Leon, and Jay Logue (and others listed in [15]) for gathering the trace, and David Surovell and Jake Brutlag for describing how WebTV client caches work. HP Labs gave us access to the Compaq trace and generous equipment support. Glenn Cooper (HP Labs) and Jeff Kopmanis (U. Michigan AI Lab) provided computer support.

We would especially like to thank Alex Rousskov, for spending late nights adding aliased-content support to Polygraph, and for answering many novice questions. Duane Wessels helped us with Squid. Mikhail Mikhailov lent us a server that made our WAN tests possible, configured it to order, and answered numerous questions. Anja Feldmann, James Hall, and Kevin Jeffay each provided information on request sizes. Fred Douglis, the reviewers from both USITS and NSDI, and our shepherd Geoff Voelker provided helpful comments.

References

[1] H. Bahn, H. Lee, S. H. Noh, S. L. Min, and K. Koh. Replica-aware caching for Web proxies. *Computer Communications*, 25(3):183–188, Feb. 2002.

[2] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Proc. 6th WWW Conf.*, Apr. 1997.

[3] cellular-news. GPRS architecture tariffs. `http://www.cellular-news.com/gprs/tariffs.php`, 2002. Tariff data seems to have disappeared from this page since 2002.

[4] A. Clark. Optimising the Web for a GPRS link. Undergraduate dissertation, Univ. of Cambridge, 2002.

[5] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: A live study of the World Wide Web. In *Proc. 1st USITS*, pages 147–158, Dec. 1997.

[6] A. Feldmann, J. Rexford, and R. Caceres. Efficient poli-

cies for carrying Web traffic over flow-switched networks. *IEEE/ACM Trans. Networking*, 6(6):673–685, Dec. 1998.

[7] E. W. Felten, Sept. 2003. Personal communication.

[8] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In *Proc. of 7th ACM Conference on Computer and Communications Security*, Nov. 2000.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol—HTTP/1.1, June 1999.

[10] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel. Reducing the energy usage of office applications. In *Proc. IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware 2001)*, pages 252–272, Heidelberg, Germany, Nov. 2001.

[11] B. Hannigan, C. D. Howe, S. Chan, and T. Buss. Why caching matters. Technical report, Forrester Research, Inc., Oct. 1997.

[12] V. Henson. An analysis of compare-by-hash. In *Proc. HotOS IX*, Lihue, HI, May 2003.

[13] F. Hernandez-Campos, K. Jeffay, and F. Smith. Tracking the evolution of Web traffic: 1995-2003. In *Proc. MASCOTS*, Orlando, FL, Oct. 2003.

[14] T. Kelly. *Optimization in Web Caching*. PhD thesis, University of Michigan, July 2002.

[15] T. Kelly. Thin-client Web access patterns: Measurements from a cache-busting proxy. *Computer Communications*, 25(4):357–366, Mar. 2002.

[16] T. Kelly and J. Mogul. Aliasing on the World Wide Web: Prevalence and performance implications. In *Proc. 11th Intl. World Wide Web Conf.*, pages 281–292, Honolulu, HI, May 2002.

[17] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance issues of enterprise level Web proxies. In *Proc. SIGMETRICS*, pages 13–23, Seattle, WA, June 1997.

[18] P. Mattis, J. Plevyak, M. Haines, A. Beguelin, B. Totty, and D. Gourley. U.S. Patent #6,292,880: "Alias-free content-indexed object cache", Sept. 2001.

[19] J. C. Mogul. Squeezing more bits out of HTTP caches. *IEEE Network*, 14(3):6–14, May/June 2000.

[20] J. C. Mogul and A. V. Hoff. Instance digests in HTTP. RFC 3230, IETF, Jan. 2002.

[21] J. C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, IETF, Jan. 2002.

[22] D. Mosberger and T. Jin. `httperf`: A tool for measuring Web server performance. In *Proc. First Workshop on Internet Server Performance*, pages 59–67, Madison, WI, June 1998.

[23] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th SOSP*, pages 174–187, Oct. 2001.

[24] National Institute of Standards and Technology. Secure hash standard. FIPS Pub. 180-1, U.S. Dept. of Commerce, Apr. 1995. `http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt`.

[25] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. ACM SIGCOMM*, pages 155–166, Sept. 1997.

[26] V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. In *Proc. 2nd WWW Conf.*, pages 995–1005, Chicago, IL, Oct. 1994.

[27] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison Wesley, Dec. 2001.

[28] S. C. Rhea, K. Liang, and E. Brewer. Value-based Web caching. In *Proc. WWW 2003*, pages 619–628, Budapest, May 2003.

[29] R. L. Rivest. RFC 1321: The MD5 message-digest algorithm, Apr. 1992.

[30] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, 1997.

[31] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. *RSA Labs Bulletin*, 4(12):1–6, Nov. 1996.

[32] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *Proc. USENIX Annual Technical Conf.*, June 1998.

[33] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, pages 87–95, Aug. 2000.

[34] Squid Team. Squid Web proxy cache, Aug. 2002. `http://www.squid-cache.org/`.

[35] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. ISBN 0-13-588187-0.

[36] J. Touch. Performance analysis of MD5. In *Proc. SIGCOMM*, pages 77–86, Cambridge, MA, Aug. 1995.

[37] A. Tridgell and P. Mackerras. The `rsync` algorithm. Technical Report TR-CS-96-05, Dept. of Computer Science, Australian National University, June 1996.

[38] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP distribution and replication protocol. Technical Report NOTE-DRP, World Wide Web Consortium, Aug. 1997.

[39] D. Wallach, Sept. 2003. Personal communication.

[40] D. Wessels. *Web Caching*. O'Reilly, June 2001.

[41] C. E. Wills and M. Mikhailov. Examining the cacheability of user-requested Web resources. In *Proc. 4th Web Caching Workshop*, Apr. 1999.

[42] C. E. Wills and M. Mikhailov. Towards a better understanding of Web resources and server responses for improved caching. In *Proc. 8th WWW Conf.*, May 1999.

Notes

[1]In some rare cases, the client may only want to render a well-defined sub-part, such as a chapter of a PDF file.

[2]The "proceed" model, described in Section 5.1, also supports end-to-end use.

[3]We based this conclusion on other traces, because the Compaq trace does not include this data, and the header-size data in the WebTV trace appears to be unreliable, possibly the result of incorrect logic for recording header lengths in the trace-gathering process.

[4]Most Web responses are at the low end of this range; we previously summarized results from several traces showing mean sizes between 6,054B and 21,568B, and medians between 1,821 and 4,346B [16].

[5]We thank Flavia Peligrinelli Ribeiro for pointing out this attack. As far as we can determine [7,39], this form of attack has not previously been reported in the literature.

OSPF Monitoring: Architecture, Design and Deployment Experience

Aman Shaikh
AT&T Labs - Research
Florham Park, NJ 07932, USA
ashaikh@research.att.com

Albert Greenberg
AT&T Labs - Research
Florham Park, NJ 07932, USA
albert@research.att.com

Abstract

Improving IP control plane (routing) robustness is critical to the creation of reliable and stable IP services. Yet very few tools exist for effective IP route monitoring and management. We describe the architecture, design and deployment of a monitoring system for OSPF, an IP intra-domain routing protocol in wide use. The architecture has three components, separating the capture of raw LSAs (Link State Advertisements – OSPF updates), the real-time analysis of the LSA stream for problem detection, and the off-line analysis of OSPF behavior. By speaking "just enough" OSPF, the monitor gains full visibility of LSAs, while remaining totally passive and visible only at the point of attachment. We describe a methodology that allows efficient real-time detection of changes to the OSPF network topology, flapping network elements, LSA storms and anomalous behavior. The real-time analysis capabilities facilitate generation of alerts that operators can use to identify and troubleshoot problems. A flexible and efficient toolkit provides capabilities for off-line analysis of LSA archives. The toolkit enables post-mortem analysis of problems, what-if analysis that can aid in maintenance, planning, and deployment of new services, and overall understanding of OSPF behavior in large networks. We describe our experiences in deploying the OSPF monitor in a large operational ISP backbone and in a large enterprise network, as well as several examples that illustrate the effectiveness of the monitor in tracking changes to the network topology, equipment problems and routing anomalies.

1 Introduction

Effective management and operation of IP routing infrastructure requires sound monitoring systems. With the advent of applications that require a high degree of performance and stability, such as VoIP and distributed gaming, network operators are now paying considerable attention to the performance of the routing infrastructure – its convergence, stability, reliability and scalability properties. Yet very few monitoring tools exist for effective routing management and operation. In this paper we present a monitoring system for one of the widely used intra-domain routing protocols, OSPF [1] by providing its detailed architecture and design. The OSPF Monitor has been deployed in two operational networks: a large enterprise network and an ISP network. It has proved to be a valuable asset in both networks. We provide several examples illustrating different ways in which the monitor has been used, as well as the lessons learned through these experiences.

We designed the OSPF Monitor to meet the following objectives:

1. **Provide real-time tracking of OSPF behavior.** Such real-time tracking can be used for (a) identifying problems in the network and helping operators troubleshoot them, (b) validation of OSPF configuration changes made for maintenance or traffic engineering purposes, and (c) real-time presentation of accurate views of the OSPF network topology.
2. **Facilitate off-line, in-depth analysis of OSPF behavior.** Such off-line analysis can be used for (a) post-mortem analysis of recurring problems, (b) generating statistics and reports about network performance, (c) identifying anomaly signatures and using these signatures to predict impending problems, (d) tuning configurable parameters, and (e) improving maintenance procedures.

There are two basic approaches for monitoring OSPF: rely on SNMP [2] MIBs and traps, or listen to Link State Advertisements (LSAs) flooded by OSPF to describe the network changes. Our prior work [3] has shown the superiority of the LSA-based approach, so we take the approach of passively listening to LSAs for our OSPF Monitor. The monitor directly attaches to the network, and speaks enough OSPF to receive LSAs. These LSAs are then analyzed in real-time to identify network problems and validate configuration changes. LSAs are also archived for a detailed off-line analysis, for example, for identification and diagnosis of recurring problems. The monitor uses a three-component architecture to provide a stable, scalable and flexible solution. The three components are:

1. *LSA Reflector (LSAR)* which collects LSAs from the network,
2. *LSA aGgregator (LSAG)* which analyzes LSA streams in real-time to identify problems, and
3. *OSPFScan* which provides off-line analysis capabilities on top of LSA archives.

The paper describes these three components in detail and the benefits offered by this three-component architecture. Since the LSAR and LSAG are key to real-time monitoring, their efficiency and scalability are of utmost importance. We demonstrate the efficiency and scalability of the LSAR and LSAG in terms of network size and LSA rate through lab experiments.

This paper is organized as follows. We discuss related work in Section 2. Section 3 provides an overview of OSPF. Section 4 discusses the three-component architecture of the OSPF Monitor. Sections 5, 6 and 7 provide detailed description of these three components. Section 8 presents the performance analysis of LSAR and LSAG through lab experiments. In Section 9, we describe salient aspects of our experiences with deploying the monitor in commercial networks. Finally, Section 10 presents conclusions.

2 Related Work

Monitoring and analyzing dynamics of routing protocols have become active areas of research of late. Route monitoring systems have started to appear in the market-place from networking startups, such as Packet Design [4] and Ipsum Networks [5]. However, the products offered by these companies have appeared in the market after our OSPF Monitor was designed. Moreover, details about the architecture and implementation of these products are not available in the public domain. The IP monitoring project at Sprint [6] consists of an IS-IS listener and a BGP listener that collects IS-IS and BGP data from the Sprint network. Although a number of studies have appeared based on the data collected by these listeners, the actual architecture of the monitoring system has not received attention. Our prior work [7] and Watson *et al.* [8] presented case studies of OSPF dynamics in real networks. Although [7] used the OSPF Monitor described in this paper to collect and analyze the OSPF data for the case study, the paper did not focus on the design and implementation of the monitor itself. Neither did [8] focus on the design of the monitor. Route-Views [9] and RIPE [10] collect and archive BGP updates from several vantage-points; a number of research studies have benefited from this data. However, both Route-Views and RIPE merely collect BGP updates; they do not provide software for monitoring or analyzing the updates.

Recall that one of the design goals of the OSPF Monitor is to track the OSPF topology. Several studies have dealt with the discovery and tracking of the network topology. For instance, our prior work [3] described SNMP and LSA-based approaches for designing an OSPF topology server, and evaluation of these approaches in terms of operational complexity, reliability and timeliness of information. The evaluation showed the superiority of the LSA-based approach in terms of reliability and robustness over the SNMP-based approach. This paper extends the LSA-based approach for monitoring OSPF. The Rocketfuel project [11, 12, 13] tackled the problem of inferring ISP topologies and weight settings through end-to-end measurements. Feldmann *et al.* [14] described the approach of periodically dumping router configuration files of routers. This approach provides a static view of the topology. One can make it more dynamic by increasing the dumping frequency, but it is hard to go beyond certain limits because of the size of IP networks today. Lakshman *et al.* [15] mentioned approaches for real-time discovery of topology in their work on the RATES System for MPLS traffic engineering. But topology discovery was just one of the modules of their system and they did not go into details. Siamwalla *et al.* [16] and Govindan [17] discussed topology discovery methods that do not require cooperation from the network service providers, relying on a variety of probes, including pings and traceroutes. Such methods provide indications of interface up/down status and router connectivity. However, these methods do not deal directly with OSPF topology tracking or monitoring, the topic of this paper.

3 OSPF Overview

OSPF [1] is a link state routing protocol. With link state protocols, each router within the domain discovers and builds a complete and consistent view of the network topology as a directed graph. Each router represents a node in the graph, and each link between neighboring routers represents a unidirectional edge. Each link also has an associated weight that is administratively assigned in the configuration file of the router. Using the weighted topology graph, each router computes a shortest path tree with itself as the root, and applies the results to build its forwarding table. This assures that packets are forwarded along the shortest paths defined in terms of link weights to their destinations. We will refer to the computation of the shortest path tree as an *SPF computation*, and the resultant tree as an *SPT*.

For scalability, an OSPF domain may be divided into areas determining a two-level hierarchy. Area 0, known as the *backbone area*, resides at the top level of the hierarchy and provides connectivity to the non-backbone areas (numbered 1, 2, ...). OSPF assigns each link to exactly one area. The routers that have links to multiple areas are called *border routers*. Every router maintains a separate copy of the topology graph for each area it is connected to. In general, a router does not learn the entire topology of remote areas (i.e., the areas in which the router does not have links), but instead learns the weight of the shortest paths from one or more border routers to each node in remote areas. In addition, the reachability of external IP prefixes (associated with nodes outside the OSPF domain) can be injected into OSPF. Roughly, reachability to an external prefix is determined as if the prefix were a node linked to the router that injects the prefix into OSPF.

Routers running OSPF describe their local connectivity in *Link State Advertisements (LSAs)*. These LSAs are *flooded* reliably to other routers in the network, which the routers use to build the consistent view of the topology described earlier. The set of LSAs in a router's memory is called the *link state database* and conceptually forms the topology graph for the router. A change in the network topology requires affected routers to originate and flood appropriate LSAs. For instance, when a link between two routers comes up, the two ends have to originate and flood LSAs describing the new link. Moreover, OSPF employs a periodic refresh of LSAs. The default value of the refresh-period is 30 minutes. So, even in the absence of any topological changes every router has to periodically flood self-originated LSAs. Due to the reliable flooding of LSAs, a router can receive multiple copies of a change or refresh triggered LSA. We term the first copy received at a router as *new* and subsequently received copies as *duplicates*.

Two routers are termed neighbor routers if they have interfaces to a common network (i.e, they have a link-level connectivity). Neighbor routers form an *adjacency* so that they can exchange routing information with each other. OSPF allows a link between the neighbor routers to be used for forwarding only if these routers have the same view of the topology, i.e., the same link state database. This ensures that forwarding data packets over the link does not create loops.

4 Architecture

As mentioned earlier, the OSPF Monitor consists of three components:

1. **LSA Reflector (LSAR):** The LSAR captures LSAs from the network. Section 5 describes various modes used by the LSAR for network attachment. The LSAR sends the LSAs over a TCP connection to the real-time analysis component, and also archives them for off-line analysis.
2. **LSA aGgregator (LSAG):** The LSAG receives a stream of LSAs from one or more LSARs, and performs real-time analysis of the stream. The LSAG maintains and populates a model of the OSPF network topology (as described in Section 6), using the LSA stream.
3. **OSPFScan:** The OSPFScan is used for off-line analysis of the LSA archives. The OSPF-Scan implements a three step analysis method, described in Section 7.

Figure 1 depicts how the three components are deployed in an example OSPF network.

Separating real-time monitoring into the LSAR and LSAG components provides several benefits. Each function is simplified and can be replicated independently to increase the overall reliability. Another benefit is that the LSAG can selectively receive a subset of LSAs, for example, LSAs belonging to a given OSPF area. Furthermore, the LSAR has to reside close to the network to capture LSAs, and so must be very simple in order to achieve a high degree of reliability. Moreover, as shown in Figure 1, multiple LSAR boxes may be required to cover all

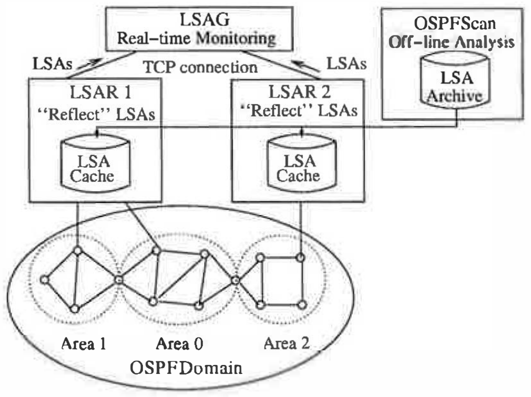


Figure 1: Three component architecture of the OSPF Monitor.

the areas since most LSAs only have an area-level flooding scope. Multiple LSAR boxes becomes almost a necessity if areas are geographically widespread. Finally, the LSAG, having to support applications, may require more complex processing and more frequent upgrades. Separating the LSAG from the LSAR allows us to bring LSAG up and down without disturbing LSARs.

Separating the real-time analysis (LSAG) from the off-line analysis (OSPFScan) offers a number of benefits. The LSAG, being real-time, needs to be very reliable (24x7 availability) and efficient. This requires us to be extremely careful about what analysis capabilities are supported by the LSAG. The OSPFScan, on the other hand, is required to process a large volume of data as efficiently as possible and allow users to query the archives. However, it also has freedom in terms of what analysis capabilities it can support. Although, real-time and off-line analysis are implemented as separate components, they work hand in hand. Any analysis capability that is supported in real-time is also supported as an off-line playback.

5 LSAR

The LSAR captures LSAs from the network for real-time and off-line analysis. LSA traffic is reliably flooded by OSPF, not routed since LSAs need to be reliably communicated even when routing is impaired or broken. As a result, the LSAR has to be closely attached to the network it monitors. There are four choices for network attachment. Below, we describe these four choices along with their pros and cons:

1. **Wire-tap mode:** An obvious way of capturing LSAs from a network is to use a tap on a link of the network, either a physical tap or port forwarding on a layer-2 switch. We will generically refer to this way of capturing LSAs as the *wire-tap* mode. If done in the right manner, this allows one to capture LSAs in a completely passive manner. However, depending on the physical media or the layer-2 technology being used, wire-tapping may not be operationally feasible. As a result, the LSAR currently does not support wire-tap, though it should be relatively straight-forward to add a module that supports LSA capture via wire-tap.
2. **Host mode:** LSAs are exchanged via a multicast group *all-rtrs* on a broadcast network [1]. Thus, on broadcast networks, LSAs can be received by joining this group; we term this mode of network attachment as the *host mode*. In this mode, the LSAR does not have to establish any form of adjacency with operational routers in the network. As a result, the LSAR is completely invisible to the routers, which is the ideal situation for any passive monitoring system. However, there are a few disadvantages. First, the LSAR has to initialize the database as routers flood LSAs on the network during the refresh process. In the worst case, it can take one refresh cycle (30 minutes as per [1]) for the LSAR to receive the first copy of an LSA after it comes up. Second, OSPF's reliable flooding does not extend to the LSAR in the host mode. If the LSAR misses transmission of an LSA to the multicast group for any reason, the sending router will be unaware of this, and there will be no retransmission. Finally, the host mode can be used only on a broadcast capable media where LSAs are sent to the multicast group.
3. **Full adjacency mode:** On a point-to-point link where routers do not send LSAs to the multicast group, the LSAR has to establish an adjacency with a router to receive LSAs. We will refer to this mode as the *full adjacency mode*. In this mode, the LSAR cannot be completely invisible to the network. However, it is crucial to ensure that the LSAR has minimal impact on the network, and most importantly, other routers in the network never send data packets to the LSAR to be forwarded elsewhere. A natural line of defense is to use router configuration measures: assign high OSPF weights and install strict access control lists and route filters on the link to the LSAR. Another line of defense stems from the fact that the LSAR (by design) cannot send LSAs. As a result, the link from the LSAR to the

router it attaches to does not exist in the OSPF topology graph. Since OSPF uses a link for data forwarding only if both of its unidirectional edges exist in the graph [1], this ensures that the LSAR-router link cannot be used for data forwarding. However, the LSAR might still have an impact on the network since the router advertises a link to the monitor in its router LSA. If the LSAR or its link with the router is flapping (going up and down), the associated adjacency can start flapping, triggering SPF calculations around the network.

4. **Partial adjacency mode:** To prevent network-wide SPF calculations when the adjacency between the LSAR and the router is flapping in the full adjacency mode, we can keep the adjacency in an "intermediate" state at the router, so that the router does not include a link to the LSAR but still sends LSAs to it. We refer to this mode as the *partial adjacency mode*. To keep the LSAR-router adjacency in the intermediate state, the LSAR describes a "fake" LSA to the router during the link state database synchronization process but never actually sends it out to the router. As a result, the database is never synchronized, the adjacency stays in OSPF's *loading* state, and is never fully established. Note that this is permissible under the OSPF specification [1]. With the partial adjacency, instability at the LSAR does not impact other routers in the network, which makes for an attractive choice. However, there are two potential issues. First, with an adjacency in the intermediate state, the router cannot delete LSAs from its link state database [1]. In the worst case, this might lead to memory exhaustion on the router. We deal with this problem by periodically dropping the partial adjacency (so that the router has a chance of garbage collecting the link state database) and then re-establishing the partial adjacency after a short time interval. In our implementation, the LSAR drops its adjacency once every 24 hours for a five second period. While there is a possibility for the LSAR to lose data during this five second period, we believe that chances of this happening are rare. Second, the use of the partial adjacency is a deviation from the normal behavior of OSPF. Keeping an adjacency in the *loading* state on a router for a long time might generate alarms, or might cause the router to drop the adjacency. However, we have not observed this problem with the commercial routers we have tested.

6 LSAG

As mentioned in Section 4, the LSAG receives a stream of LSAs from the LSAR for real-time analysis. The LSAG prints messages on the console when it detects changes to the network topology or a behavior that does not conform to the OSPF standards. These messages allow operators to identify problems in the network. Section 6.1 provides more details about various types of messages generated by the LSAG. Under the hood, the LSAG maintains and updates a snapshot of the network topology to identify topological changes and anomalous behavior. Section 6.2 provides a detailed description of the model and how it helps LSAG print console messages. This topology model also allows the LSAG to dump topology snapshots periodically and upon topological changes. These snapshots in turn can be used by applications that might benefit from them.

6.1 Classification of Real-time Messages

(1) Topology Change Messages

- **(a) Change messages insides an area**
  - **(i) Messages about a router**
    - RTRUP
    - RTR DOWN
    - BECAME BORDER RTR
    - NO LONGER BORDER RTR
    - BECAME ASBR
    - NO LONGER ASBR
  - **(ii) Messages about an interface on a router**
    - INTFDOWN
    - INTF MASK CHANGE
  - **(iii) Messages about an adjacency**
    - ADJACENCY UP
    - ADJACENCY DOWN
    - ADJACENCY COST CHANGE
    - ALL ADJACENCIES DOWN
  - **(iv) Messages about a host–route on a router**
    - STUB LINK UP
    - STUB LINK DOWN
    - STUB LINK COST CHANGE
  - **(v) Messages about DR on a broadcast network**
    - NEW DR
    - NO LONGER DR
    - DR CHANGE
    - MASK CHANGE
- **(b) Change messages for remote areas**
  - **(i) Messages about a prefix in a remote area**
    - TYPE–3 ROUTE ANNOUNCED
    - TYPE–3 ROUTE WITHDRAWN
    - TYPE–3 ROUTE COST CHANGE
  - **(ii) Messages about an ASBR in a remote area**
    - TYPE–4 ROUTE ANNOUNCED
    - TYPE–4 ROUTE WITHDRAWN
    - TYPE–4 ROUTE COST CHANGE
- **(c) Change messages for external routes**
  - TYPE–5 ROUTE ANNOUNCED
  - TYPE–5 ROUTE WITHDRAWN
  - TYPE–5 ROUTE COST CHANGE
  - TYPE–5 ROUTE COST_TYPE CHANGE
  - TYPE–5 ROUTE FORW_ADDR CHANGE

(2) Flap Messages

- RTR FLAP
- INTFFLAP
- ADJACENCY FLAP
- STUB LINK FLAP
- TYPE–3 ROUTE FLAP
- TYPE–4 ROUTE FLAP
- TYPE–5 ROUTE FLAP

(3) Messages related to Anomalous Behavior

- NON–BORDER RTR YET TO WITHDRAW TYPE–3/4 ROUTES
- NON–ASBR YET TO WITHDRAW TYPE–5 ROUTES
- DUPLICATE ADJACENCY
- DUPLICATE STUB LINK
- TYPE–3 ROUTE FROM NON–BORDER RTR
- TYPE–4 ROUTE FROM NON–BORDER RTR
- TYPE–5 ROUTE FROM NON–ASBR

(4) LSA Storm Messages

- LSA STORM

Figure 2 shows classification of various message types supported by the LSAG. Each message contains a time-stamp, and a message type along with the attributes. The message type is used for identifying the kind of event or problem in the network. For example, a "RTR DOWN" message is issued when OSPF process on a routes dies. The attributes provide more detail about the message. For example, if the message type is "RTR DOWN", the attributes include the router-id of the associated router.

Let us describe the four top-level categories shown in Figure 2:

1. **Topology Change Messages:** These messages are generated for changes in the network topology. As Figure 2 shows, majority of message types fall into this category. These messages help operators identify problems in the network, as well as help them validate maintenance activities.
2. **Flap Messages:** These messages are generated for network elements (e.g., router, link etc.) that go up and down repeatedly. These messages are always preceded by topology change messages. For example, a "RTR FLAP" message is preceded by several "RTR DOWN" and "RTR UP" messages. These messages are useful for getting an operator's attention. They also act as early warning signs to the network stability.
3. **Messages related to Anomalous Behavior:** These messages are generated when the observed behavior deviates from the expected behavior of OSPF. An example of an anomalous behavior is a non-border router originating summary LSAs (the corresponding message type is "TYPE-3 ROUTE FROM NON-BORDER RTR"). Often these messages indicate configuration errors or bugs in the vendor software.
4. **LSA Storm Messages:** These messages are generated when too many refresh instances of an LSA are observed by the LSAG. Often these messages indicate bugs in the vendor software. They also act as early warning signs to the network stability.

6.2 Topology Model

The LSAG uses a topology model to generate messages described in the previous section. Apart from generating these messages, the LSAG is also required to efficiently support real-time queries about the network topology, such as "how many routers does area X have?" or "how many interfaces does router X have in area Y?". This requires a model that can be updated and searched efficiently, and can be scaled to networks consisting of hundreds of routers and thousands of links.

We have designed and implemented a model meeting these goals, which consists of the following six classes:

1. *Area*: represents an OSPF area.
2. *Rtr*: represents a router.
3. *AreaRtr*: represents area-specific parameters of a router. Recall that OSPF does not assign a router to an area. It assigns each interface of a router to an area. Thus, a router can have some interfaces in each of several areas. An *AreaRtr* object contains the set of interfaces a router has in a given area.
4. *Ntw*: represents a subnet or a prefix.
5. *Intf*: represents an interface of a router.
6. *Link*: represents a unidirectional weighted edge in the topology. The monitor classifies links into three types: intra-area links that represent link between a pair of routers or a router-subnet pair, inter-area links to represent summary routes injected by border routers, and external links to represent external routes redistributed into OSPF. The local and remote ends of each link are objects of one of the above mentioned five classes.

Figure 3 shows the model in terms of "containment" relationship between objects of these six classes. For example, at the highest level, the model consists of a set of *Area* objects representing the areas of the OSPF network. Each *Area* object in turn contains a set of *Ntw* objects representing all the subnets of the associated area. *AreaRtr* objects are special. Since each *AreaRtr* object represents area-specific parameters of a router, it is contained in two objects: a *Rtr* object, and an *Area* object. In our implementation, search, add and delete operations on the objects are implemented via hash tables, enabling scaling to large networks.

Upon receiving an LSA, the LSAG updates the relevant part of the topology model. As an example, consider what happens when the LSAG receives a router LSA. The LSAG has to update the *AreaRtr* object that represents the router LSA. This may result in addition or deletion of interfaces, or a change in the administrative weight of interfaces. Router LSA can also result in addition

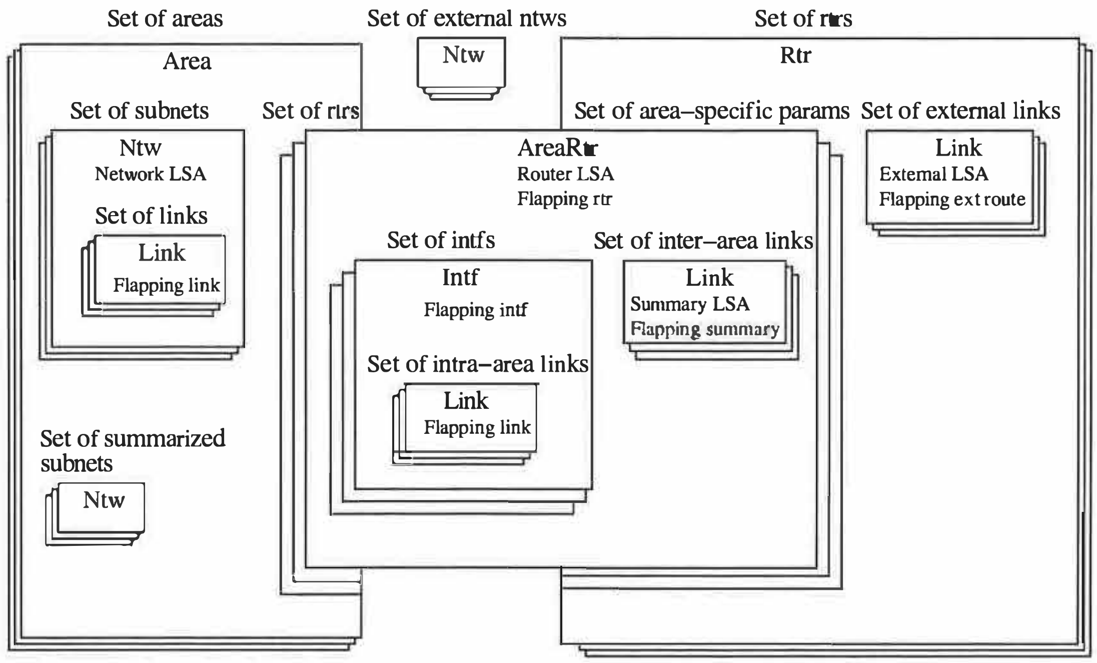


Figure 3: Topology model of the LSAG.

or deletion of an *AreaRtr* object.

The topology model conveniently allows the identification of a flapping node and generation of the associated flap message (see Section 6.1). The LSAG considers a node to be flapping if the node goes down and comes up n times within a t second time frame, where both n and t are configurable parameters. To identify flaps, each node of the OSPF topology is mapped to a specific object of the topology model. This mapping is shown in Figure 3 by statements of the form "flapping xxx" under appropriate objects. For example, an external route is mapped to a *Link* object in the set of external links in the *Rtr* object representing the advertising router.

The topology model is also used for identifying LSA storms. As described in Section 6.1, the LSAG issues an LSA storm message if many new copies of a refresh LSA are received within a short time period. Note that LSAs that indicate change(s) to the model are not considered a part of an LSA storm. More precisely, the LSAG issues a warning about too many LSA copies if it receives n refresh copies of an LSA within t seconds. To identify LSA storms, each LSA is mapped to a specific object of the topology model. The mapping is intuitive in the sense that the LSA essentially describes the status of the object it is mapped to. For example, each router LSA is mapped to an *AreaRtr* object. Similarly, each external LSA is mapped to a *Link* object in the set of external links of a *Rtr* object.

7 OSPFScan

As mentioned in Section 4, the OSPFScan is used for off-line analysis of LSA archives. At present, the OSPFScan provides the following functionalities:

1. **Classification of LSA traffic.** The OSPFScan allows various ways of "slicing-and-dicing" of LSA archives. For example, it allows isolating LSAs indicating changes from the background refresh traffic. As another example, it also allows classification of LSAs (both change and refresh) into new and duplicate instances. We have used this capability of the OSPFScan to analyze one month worth of LSA traffic as a case study for the enterprise network [7].
2. **Modeling topology changes.** Recall that OSPF represents the network topology as a graph. Therefore, the OSPFScan allows modeling of OSPF dynamics as a sequence of changes to the underlying graph where a change represents addition/deletion of vertices/edges to this graph. Furthermore, the OSPFScan allows a user to analyze these changes by saving each change as a single topology change record. Each such record contains information about the topological element (vertex/edge) that changed along with the nature of the change. For example, a router is treated as a vertex, and the record contains the

OSPF router-id to identify it. As another example, a link between a pair of routers is treated as an edge, and the corresponding record uses router-ids of the two ends to identify the link. We have used change records for a detailed analysis of router/link availability as we will see in Section 9.1.2.

3. **Emulation of OSPF routing.** The OSPFScan allows a user to reconstruct the routing table of any given set of routers at a given point of time based on the LSA archives. For a sequence of topology changes, the OSPFScan also allows the user to determine changes to these routing tables. Together, these capabilities allow the user to determine an end-to-end path through the OSPF domain at a given time, and see how this path changed in response to network events over a period of time.
4. **Statistics and reports.** The OSPFScan allows generation of statistics and reports on specific OSPF dynamics and anomalies over given time intervals. A simple example is the ability to count the number of change, new and duplicate LSAs over a given time period.
5. **Correlation with other data sources.** The functionalities provided by the OSPFScan form a basis for correlating OSPF data with other data sources such as usage data (e.g., SNMP statistics and Cisco netflow statistics), fault data (e.g., SNMP traps and syslogs), network inventory and topology data (e.g., router configuration files), other dynamic routing data (e.g., BGP updates), and maintenance data (workflow logs). For example, the routing table entries generated by the OSPFScan have been used by Teixeira *et al.* [18] to analyze the impact of OSPF changes on BGP routing.

The OSPFScan implements a three-step procedure to analyze each LSA record. These three steps include parsing the LSA, testing the LSA against a query expression, and analyzing the LSA if it satisfies the query. The OSPFScan allows a user to specify the query expression and the kind of analysis to be carried out with the LSAs.

The parsing step converts each LSA record of the archive into a *canonical form*. The query expression is applied to the canonical form, and not to the raw LSA record. The use of a canonical form makes it easy to adapt OSPFScan's functionality to support LSA archive formats other than the native format used by the LSAR. Adaptation only requires addition of a routine to parse the new format into the canonical form. The query language supported by the OSPFScan has a C-style expression syntax. An example query expression is "areaid == '0.0.0.0'" which selects all the LSAs belonging to area 0. The OSPFScan uses an internally developed data stream scan library which allows efficient processing of arbitrary data, described via a canonical form for each data type. The OSPFScan also allows further analysis of the information derived from the LSA archives such as topology changes and routing entries by implementing a similar three-step procedure.

8 Performance Evaluation

In this section, we characterize the performance of the monitor through lab experiments. We focus on the LSAR and LSAG which are central to the real-time monitoring, and analyze how these two components scale with the LSA-rate and the network size.

8.1 Methodology

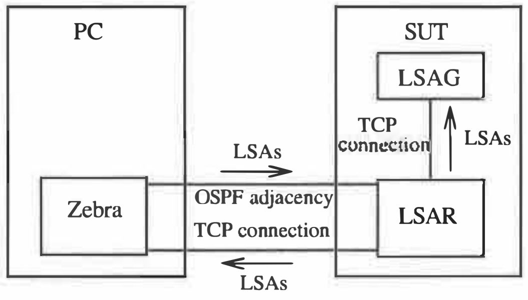


Figure 4: Experimental setup for measuring the LSAR and LSAG performance.

Our experimental setup consists of two hosts as shown in Figure 4. The host denoted by *SUT (System Under Test)* runs the LSAR and LSAG. The other host runs a modified version of Zebra [19]. The modifications include the ability to emulate a desired OSPF topology and changes to it by sending appropriate LSAs over an OSPF adjacency, and the ability to form an LSAG session with the LSAR to receive LSAs.

With this setup, we start an experiment by loading the desired topology into the LSAR running on the SUT. We use a fully connected graph having n nodes as the emulated topology. Once the desired topology is loaded at the LSAR, Zebra sends out a burst of back-to-back LSAs to the LSAR; we will denote the number of LSAs in a burst by l. These bursts are repeated such that there is a gap of inter-burst time (i) between the beginning of successive bursts. Thus, every experiment instance consists of four input parameters: the number of nodes (n) in the fully connected graph, the number of LSAs in a burst (l), inter-burst time (i), and the number of bursts (b).

Each LSA in a burst results in changing the status of *all* n adjacencies of a router from up to down or from down to up. During a burst, we cycle through routers while sending the LSAs out. For example, if $n = 2$ and $l = 4$, the four LSAs sent out would result in the following events: (i) bring down all adjacencies of router 1, (ii) bring down all adjacencies of router 2, (iii) bring up all adjacencies of router 1, and (iv) bring up all adjacencies of router 2. We believe that using a fully connected graph, flapping adjacencies of routers, and sending out bursts of LSAs stresses the LSAR and LSAG most in terms of resources.

To characterize the LSAR performance, we measure how quickly it can send out an LSA received over an OSPF adjacency to the LSAG. Recall that our modified Zebra is capable of forming an LSAG session with the LSAR. This allows us to record the necessary time-stamps within Zebra itself thereby obviating the need for running a separate LSAG process on the Zebra PC. For each LSA, Zebra records the time when it sends the LSA over the adjacency, and the time when it receives the LSA back over the LSAG session. We will denote the mean of the difference between the send-time and receive-time for an LSA by T_{lsar}. For the LSAG, we measure how long it takes the LSAG to process every LSA. To measure this, we instrumented the LSAG code to record the time before and after every LSA is processed. We will denote the mean LSA processing time at the LSAG by T_{lsag}.

Long duration LSA bursts can cause the LSAR to lose LSA instances occasionally. Despite OSPF's reliable flooding, most of these losses are irrecoverable if the lost instance is "overwritten" by a new instance of the LSA before the retransmission timer expires. Therefore, we measure the number of LSAs lost during each experiment by calculating the fraction of LSAs that were sent by Zebra to the LSAR but never received back. We will denote this quantity by L_{lsar}.

8.2 Results

We used PCs each having a 550 MHz AMD-K6 CPU, 64 MB of RAM and RedHat Linux 6.2 as the SUT and for running Zebra. We varied the number of nodes (n) in the topology in the range $50, 60, \ldots, 100$; and varied the number of LSAs per burst (l) in the range $50, 100, \ldots, 500$. We set the inter-burst gap (i) to one second, and sent 100 bursts (b) in each experiment. For every value of (n, l, b, i) quadruplet, we carried out the experiment three times.

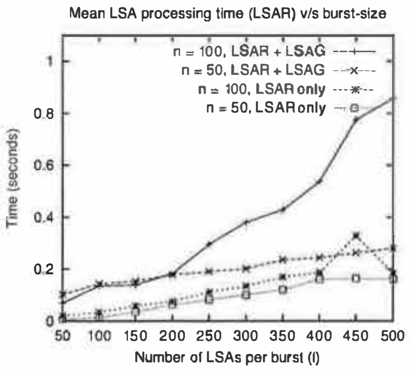


Figure 5: LSAR performance (T_{lsar}) versus the number of LSAs per burst for 50 and 100 nodes in the topology.

Figure 5 shows how T_{lsar} varies as a function of the burst-size (l) for two values of n. Apart from running both LSAR and LSAG on the SUT, we also repeated the same set of experiments with only LSAR running on the SUT. As Figure 5 shows, T_{lsar} increases as the burst-size increases under all circumstances. This can be explained as follows. The inter-departure time while sending a burst of LSAs out at Zebra is less than the inter-arrival time of receiving them back since the inter-arrival time includes the two-way propagation delay and the processing time per LSA. As a result, the turn-around time (the difference between receive-time and send-time) is lowest for the first LSA within a burst, and gradually increases for subsequent LSAs within the same burst. Moreover, this disparity in the turn-around time across LSAs widens as the burst-size increases, ultimately resulting in a higher value of T_{lsar} for higher burst-sizes. For the "LSAR only" case, the number of nodes in the topology does not have much impact on T_{lsar} since the LSAR merely deals with passing LSAs to LSAG (Zebra in this case) and archiving. On the other hand, T_{lsar} is higher for the "LSAR + LSAG" case than the "LSAR only" case and is sensitive to the number of nodes in the topology. Both these observations can be attributed to the LSAG contending for CPU and memory with the LSAR on the SUT.

Figure 6 shows how T_{lsag} varies as a function of the number of nodes (n) for two values of the burst-size. As expected, T_{lsag} increases linearly

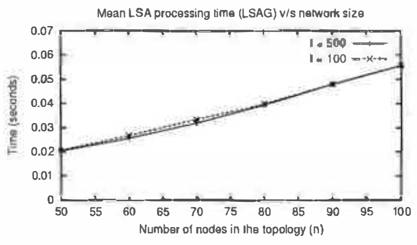


Figure 6: LSAG performance (T_{lsag}) versus the number of nodes in the network for the burst-size equal to 100 and 500 LSAs.

with the number of nodes in the topology. However, it is insensitive to the burst-size mainly because of flow control imposed by TCP.

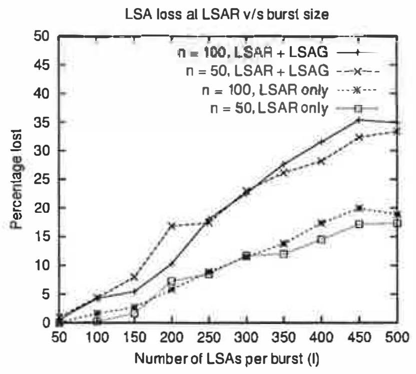


Figure 7: Loss-rate at the LSAR (L_{lsar}) versus the number of LSAs per burst for 50 and 100 nodes in the topology.

Figure 7 shows how L_{lsar} varies as a function of the burst-size for two values of n. As expected, L_{lsar} increases as the burst-size increases, and L_{lsar} is higher for the "LSAR + LSAG" case than the "LSAR only" case. However, L_{lsar} is insensitive to the topology size. For all the experiments, we checked that the LSAs not received by Zebra did not exist in the LSAR archive implying that the LSAR never received them due to a memory exhaustion in the IP stack on the SUT. Unlike LSAR, the LSAG does not lose LSAs due to the use of TCP. In fact, if given enough time at the end of the experiment, the LSAG ultimately is able to process all the LSAs sent by the LSAR.

These results show that the LSAR and LSAG are capable of handling large networks and high LSA-rate even on a low-end PC. Furthermore, their performance degrades gracefully as the load increases beyond their processing capability. We believe that at high LSA-rates and excessive flapping as was the case here the routers are more likely to melt-down before the LSAR and LSAG processes especially if a high-end server is used for running these processes. This is because route processors used on even high-end routers do not tend to be as powerful as the ones used on servers. Furthermore, route processors typically run processes other than OSPF, whereas the LSAR and LSAG can run on dedicated servers.

9 Deployment Experience

We have deployed the monitor in two commercial networks: a large enterprise network and a large ISP network. Table 1 provides some facts about these two networks and the deployment of the monitor. More details about the enterprise network architecture can also be found in [7]. It is important to note how diverse the two networks are in terms of their layer-2 architecture, and their use of OSPF. The layer-2 architecture dictated which LSAR mode we used for network attachment, whereas the use of OSPF affected the number of LSAs received per day. All servers running LSAR and LSAG processes are NTP-synchronized in both the networks.

Operators of both the networks were very cautious about deploying the LSAR and LSAG in their networks. As a result, before the deployment, both LSAR and LSAG underwent extensive testing and review to convince the operators about passiveness of the LSAR and robustness of both the components. During testing, we also ensured that there were no memory leaks in either of the components. To further enhance security, we turned off all unnecessary services, and put in measures to tightly control access to the server running the LSAR. To increase the accuracy of the time-stamping of the LSAs being archived at the LSAR, we took care of minimizing the I/O that might result from activities such as writing of syslog messages. In addition, to keep the measurements as clean as possible, we separated the interfaces used for remote management from those used to capture LSAs. This minimized the competition between measurement and management traffic.

The OSPF Monitor has handled the load imposed by these networks with ease, demonstrating and further confirming the scalability of the system. Furthermore, the LSAR and LSAG implementations have proved to be extremely reliable and robust. We have observed the LSAR crash only once during these two years, and have not observed the LSAG crash at all. The LSAR

Parameter	**Enterprise network**	**ISP network**
Layer-2 architecture	Ethernet-based LAN	Point-to-point links
Customer reachability information	Use of EIGRP between network-edge routers and customer-premise routers. EIGRP routes are imported into OSPF.	Use of I-BGP to propagate customer reachability information. I-BGP routes are not imported into OSPF.
Scale of monitor deployment	15 areas; 500+ routers	Area 0; 100+ routers
LSAR attachment mode	Host mode	Partial adjacency mode
Deployment history	Deployment started in February, 2002 by connecting LSAR to two areas. Remaining areas were gradually covered in the next two months.	Deployment started in January, 2003 by connecting LSAR to area 0.
Size of the LSA archive	10 MB per day	8 MB per day

Table 1: Facts related to the deployment of the OSPF monitor in two commercial networks.

has been upgraded three times for bug fixes, and eight times for enhancements, whereas the LSAG has been upgraded three times for bug fixes and 26 times for enhancements.

9.1 Utility of the Monitor

The OSPF monitor is being used primarily in two ways. First, the LSAG is used for day-to-day network operation, continuously tracking the health of the network. Second, the OSPFScan is used for detailed and long term analysis of OSPF behavior.

9.1.1 LSAG in Day-to-day Operations

As mentioned earlier, the LSAG provides two data sources in real-time: messages related to the topology changes and anomalous behavior, and network topology snapshots. Both the sources provide valuable insight into the health of a network.

We have developed a web-site for viewing LSAG messages, interacting with network operators to make the site as simple and user-friendly as possible. The web-site allows the operators to query the LSAG message logs, generate statistics about the messages, and navigate past archives. The web-site makes use of a configuration management tool to map IP addresses into names. This web-site is now used extensively by network support and operations on a regular basis, and has proved invaluable during network maintenance to validate maintenance steps as well as to monitor the impact of maintenance on the network-wide behavior of OSPF.

Network operation groups also use the LSAG messages for generating alarms by feeding them into higher layer alerting systems. This in turn allows correlation and grouping with other monitoring tools. To prevent a deluge of alerts generated due to a high frequency of LSAG messages, we have taken two steps. First, we prioritize messages to help operators in the event of "too many flashing lights". For example, the alerting system assigns "RTR DOWN" message a higher priority than a "RTR UP" message. Second, we group multiple messages into a single alarm. For example, a fiber cut can bring down a number of adjacencies prompting the LSAG to generate several "ADJACENCY DOWN" messages. We group all these messages into a single alert to prevent a flurry of alerts for a single underlying event.

Network operators may change OSPF link weights from their design values to carry out maintenance tasks. We have designed a "link-audit" web-site that allows operators to keep track of such link weight changes. The web-site makes use of the topology snapshots to display the set of links whose weights differ from the design weights. This allows operators to validate the steps carried out for maintenance. At the end of the maintenance interval, the web-site also allows operators to verify that weights of the affected links are reverted back to their original values.

Below we describe a few specific cases where the LSAG served to identify network problems.

1. **Internal problem in a crucial router:** The LSAG identified an intermittent hardware problem in a crucial router in area 0 of the enterprise network [7]. This problem resulted in

episodes lasting a few minutes during which the problematic router would drop and re-establish adjacencies with other routers on the LAN. Each episode lasted only for a few minutes and there were only a few episodes each day. The data suggests that during the episodes the network was at the risk of partitioning or was in fact partitioned. During these episodes, a second router failure could have resulted in a catastrophic loss of connectivity. Fortunately, a flurry of "ADJACENCY UP" and "ADJACENCY DOWN" messages recorded by the LSAG during each episode helped operators identify the problem, and perform preventative maintenance. It is worth noting here that this problem did not manifest in other network management tools being used by the enterprise network.

2. **External link flaps:** The LSAG helped identify a flapping external link in the enterprise network [7]. One of the enterprise network routers (call it A) maintains a link to a customer premise router (call it B) over which it runs EIGRP. Router A imports EIGRP routes into OSPF as external LSAs. LSAG messages led to a closer inspection of network conditions, which revealed that the EIGRP session between A and B started flapping when the link between A and B became overloaded. This led to router A repeatedly announcing and withdrawing EIGRP prefixes via external LSAs. The flapping of the link between A and B persisted nearly every day for months between 9 PM and 3 AM. The LSAG messages ("TYPE-5 ROUTE ANNOUNCED" and "TYPE-5 ROUTE WITHDRAWN") helped network operators to identify and mitigate the problem, though they could not completely eliminate it as the operators did not have access to the customer-premise router.
3. **Router configuration problem:** In another case, the LSAG helped operators of the enterprise network identify a configuration problem: assignment of the same router-id to two routers. This error resulted in these routers repeatedly originating their router LSAs which showed up as a series of "ADJACENCY UP/DOWN" LSAG messages.
4. **Refresh LSA bug:** The LSAG helped identify a bug in the refresh algorithm of the routers from a particular vendor in the ISP network. The bug resulted in a much faster refresh of summary LSAs under certain circumstances than the RFC-mandated [1] rate of 30 minutes. The bug was identified due to the "LSA STORM" messages generated by the LSAG. At the time of writing this paper, the vendor is investigating the bug. It is worth noting that it would be impossible to catch such a bug with any other class of available network management tools.

9.1.2 Use of OSPFScan for Detailed Analysis

In this section, we touch on ways in which the OSPFScan has been used for analyzing long-term behavior of OSPF. For both the networks where the monitor is deployed, in addition to archiving all LSAs, we also archive topology snapshots and LSAG message logs. Furthermore, we use the OSPFScan to extract change LSAs, topology change records and to compute routing tables for each router, grouped by 24-hour intervals. All this data (raw and change LSAs, topology change records, routing tables, topology snapshots, and LSAG message logs) forms the data repository for the OSPFScan analysis. Although there is a redundancy (raw LSAs are sufficient to construct all other forms of data), we have found that keeping the derived data greatly assists interactive analysis of OSPF behavior. To illustrate, suppose a user is interested in analyzing how the path between two end-points evolved over time. It is much faster to automatically compute paths between two end-points using the routing table data than to construct the paths from raw LSAs.

Specific illustrations of the OSPFScan usage include:

1. **Duplicate LSA analysis:** The LSA traffic analysis in the enterprise network by the OSPFScan [7] revealed excessive duplicate LSA traffic. For some OSPF areas, the duplicate LSA traffic formed 33% of the overall LSA traffic. Subsequent analysis led to the root-cause of the excessive traffic and preventative measures, details of which can be found in [7].
2. **Change LSA statistics:** The SPF calculation on Cisco routers is paced by two timers [20]: (i) *spf-delay*, which specifies how long OSPF waits between receiving a topology change and starting an SPF computation; and (ii) *spf-holdtime*, which determines the lag between two successive SPF computations. In order to reduce OSPF convergence time, it is desirable to decrease these timers to small values; however, reducing these values too much

can lock the routers into performing excessive SPF calculations, possibly destabilizing the network. Analysis of the inter-arrival time of change LSAs in the network can help administrators configure these timers to "good" values. The network administrators of the ISP network have done precisely this. To facilitate the process, we built a web-site on top of the change LSA repository, providing statistics such as minimum, maximum, mean, standard deviation and empirical CDF of inter-arrival times of change LSAs over a given time period and for a given LSA type.

3. **Availability analysis:** Assessing reliability and availability of intra-domain routing is crucial for deploying new services and associated service assurances into the network. OSPF monitor data has proved very useful in answering questions such as: what is the mean down-time and mean service-time for links and routers in the network at the IP level? Again, we created a web-site to answer such questions for the ISP network. The site relies on the topology change records stored in the repository.
4. **Use of OSPF routing tables:** For each router, the routing table archive contains the entire history of routing tables across the measurement interval (e.g., several months or longer). This data is being used by the ISP network engineering teams to determine and analyze end-to-end paths within the network at any instance of time, to correlate OSPF routing changes with I-BGP updates seen in the network [18], and to analyze how OSPF events impact the traffic flow within the network by correlating this data with active probing.

9.2 Lessons Learned

In this section, we point out some of the lessons learned during and after the deployment of the system. The points may help in design, development and deployment of other route monitoring systems.

- **New tools reveal new failure modes.** The LSAG has allowed us to find and fix several problems in a pro-active fashion. Some of these problems would have been impossible to find with other network management tools (e.g., the refresh LSA bug).
- **Real-time alerting and off-line analysis are complementary.** Some problems such as router-bug were caught by real-time messages, whereas some other problems such as excessive duplicate LSA traffic were caught because of off-line analysis of LSAs stored over long time intervals. Finally, problems such as refresh LSA bug were identified using LSAG messages in real-time, but they also required a more detailed off-line analysis.
- **OSPF exhibits significant amount of activity.** Based on our experience, we have noticed that both the networks monitored exhibit significant amount of OSPF activity. This activity is due to maintenance tasks as well as network problems. Efficient and scalable design of the system has helped us tackle this high level of activity with relative ease.
- **Add functionality incrementally.** We have added new functionality and improved the system by close interaction with network operators. At one level, this pertained to the user interfaces. For example, it took several iterations until the operators were satisfied with LSAG message formats and could make sense of associated logs at a glance. At another level, it was important to customize and enhance value by building custom reports that reflected operational practices.
- **Archive all the LSAs.** The analysis of excessive duplicate LSAs and refresh LSA bug required archiving all the LSAs captured from the network, not just those that indicated topology changes. The volume of all OSPF LSAs is not onerous. As seen in Table 1, the volume of raw LSAs collected from each of the two networks is in the order of 10 MB per day. This makes it fairly easy to collect all the LSAs from the network, store these for a long period, as well as transfer and replicate the archives as needed.

10 Conclusion

In this paper, we have described the architecture and design of an OSPF monitor, which passively listens to LSAs flooded in the network, provides real-time alerting and reporting on network events, and allows off-line investigation and post-mortem analysis. The three main components of the system are:

- The LSAR (LSA Reflector), which captures LSAs from the network. The LSAR supports three modes by which it can be attached safely and passively to the network.
- The LSAG (LSA aGgregator), which receives "reflected" LSAs from one or more LSARs,

monitors the network in real-time for operationally meaningful events. The LSAG populates a network topology model using the LSA stream, tracking changes to the topology, and issuing associated alerts and console messages. These messages allow network operators to quickly localize and troubleshoot problems. The topology model also provides timely and accurate views of the OSPF topology to other network management applications.

- The OSPFScan, which allows efficient post-mortem analysis of LSA archives via streamlined parse, select and analyze capabilities. The OSPFScan includes a large set of libraries, implementing, in particular, playback and isolation of topology change events, generation of statistics, and construction and evolution of routing tables and end-to-end paths for every topology change event.

We demonstrated that the LSAR and LSAG can scale to large networks and large LSA rates through lab experiments. Overall, the monitor cleanly separates instrumentation, real-time processing and off-line analysis. The monitor has been successfully deployed in two commercial networks, where it has run without glitches for months. We provided several examples illustrating how operators are using the monitor to manage these networks, as well as some lessons learned from this experience.

In future, we plan to work on improving the LSAG by incorporating more intelligent grouping and prioritization of messages. We also plan to focus on correlation of OSPF data with other data sources both for better root-cause analysis of network problems and for better understanding of network-wide interaction of various protocols.

Acknowledgments

We are grateful to the operators of the ISP and the enterprise network for allowing us to deploy the OSPF Monitor. Their subsequent feedback improved the functionality of the monitor immensely. We thank Jennifer Rexford, Jay Borkenhagen, anonymous reviewers and our shepherd, Stephan Savage for their comments which helped us a lot in improving the paper. Finally, we thank Kiranmaye Sirigineni for modifications to Zebra that enabled the OSPF topology emulation.

References

[1] J. Moy, "OSPF Version 2." Request for Comments 2328, April 1998.

[2] W. Stallings, *SNMP, SNMPv2, SNMPv3 and RMON 1 and 2*. Addison-Wesley, 1999.

[3] A. Shaikh *et al.*, "An OPSF Topology Server: Design and Evaluation," *IEEE J. Selected Areas in Communications*, vol. 20, May 2002.

[4] "Route Explorer." `http://www.route-explorer.com/`.

[5] "Route Dynamics." `http://www.ipsumnetworks.com`.

[6] "IP Monitoring Project." `http://ipmon.sprint.com/`.

[7] A. Shaikh *et al.*, "A Case Study of OSPF Behavior in a Large Enterprise Network," in *Proc. ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[8] D. Watson *et al.*, "Experiences with Monitoring OSPF on a Regional Service Provider Network ," in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[9] "University of Oregon Route Views Project." `http://www.routeviews.org/`.

[10] "RIPE Network Coordination Center." `http://www.ripe.net/`.

[11] N. Spring *et al.*, "Measuring ISP Topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, 2002.

[12] R. Mahajan *et al.*, "Inferring Link Weights using End-to-End Measurements," in *Proc. ACM SIGCOMM Internet Measurement Workshop (IMW)*, 2002.

[13] "Rocketfuel: An ISP Topology Mapping Engine." `http://www.cs.washington.edu/research/networking/rocketfuel`.

[14] A. Feldmann and J. Rexford, "IP Network Configuration for Intra-domain Traffic Engineering," *IEEE Network Magazine*, September 2001.

[15] P. Aukia *et al.*, "RATES: A server for MPLS traffic engineering," *IEEE Network*, pp. 34-41, March/April 2000.

[16] R. Siamwalla *et al.*, "Discovering Internet Topology." Unpublished manuscript, `http://www.cs.cornell.edu/skeshav/papers/discovery.pdf`, July 1998.

[17] R. Govindan and H. Tangmunarunkit, "Heuristics for Internet Map Discovery," in *Proc. IEEE INFOCOM*, March 2000.

[18] R. Teixeira *et al.*, "Dynamics of Hot-Potato Routing in IP Networks," in *Proc. ACM SIGMETRICS*, June 2004.

[19] "GNU Zebra." `http://www.zebra.org`.

[20] "Configure Router Calculation Timers." `http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr/%np1\_c/1cprt1/1cospf.html\#xtocid2712621`.

OverQoS: An Overlay based Architecture for Enhancing Internet QoS

Lakshminarayanan Subramanian* Ion Stoica* Hari Balakrishnan+ Randy H. Katz*

*University of California at Berkeley
{lakme,istoica,randy}@cs.berkeley.edu

+Massachusetts Institute of Technology
hari@nms.lcs.mit.edu

Abstract

This paper describes the design, implementation, and experimental evaluation of *OverQoS*, an overlay-based architecture for enhancing the best-effort service of today's Internet. Using a *Controlled loss virtual link (CLVL)* abstraction to bound the loss rate observed by a traffic aggregate, OverQoS can provide a variety of services including: (a) smoothing packet losses; (b) prioritizing packets within an aggregate; (c) statistical loss and bandwidth guarantees.

We demonstrate the usefulness of OverQoS using two sample applications. First, *RealServer* can use OverQoS to improve the signal quality of multimedia streams by protecting more important packets at the expense of less important ones. Second, *Counterstrike*, a popular multi-player game, can use OverQoS to avoid frame drops and prevent end-hosts from getting disconnected in the presence of loss rates as high as 10%. Using a wide-area overlay testbed of 19 hosts, we show that: (a) OverQoS can simultaneously provide statistical loss guarantees of 0.1% coupled with statistcal bandwidth guarantees ranging from 100 Kbps to 2 Mbps across international links and broadband end-hosts; (b) OverQoS incurs a low bandwidth overhead (typically less than 5%) to achieve the target loss rate, and (c) the increase in the end-to-end delay is bounded by the round-trip-time along the overlay path.

1 Introduction

Over the past decade, there have been many efforts to provide QoS in the Internet. Most notably, the Intserv and Diffserv service architectures have been proposed to offer a large array of services ranging from per flow and delay guarantees to per aggregate guarantees and priority services. Despite these efforts, today's Internet still continues to provide only a best-effort service. One of the main reasons is the requirement of these proposals that all network elements between a source and a destination implement QoS mechanisms. The inherent difficulty in changing the IP infrastructure coupled with the natural lack of incentives for ISPs to coordinate their deployment has rendered this requirement infeasible, and ultimately hurt the adoption of IntServ and DiffServ.

In this paper, rather than trying to achieve traditional QoS guarantees such as the ones offered by Intserv and Diffserv, we ask the following question: *are there any meaningful QoS enhancements that can be provided in the Internet without requiring support from the IP routers*? To answer this question we turn our attention to overlay networks as an alternative for introducing new functionality that is either too cumbersome to deploy in the underlying IP infrastructure, or that requires information that is hard to obtain at the IP level. Examples of successful overlay networks include application-layer multicast [12, 21], Web content distribution networks, and resilient overlay networks (RONs) [7].

To this end, we propose *OverQoS*, an overlay based QoS architecture for enhancing Internet QoS. The key building block of OverQoS is the *controlled-loss virtual link* (CLVL) abstraction. CLVL provides statistical loss guarantees to a traffic aggregate between two overlay nodes in the face of varying network conditions. In addition, it enables overlay nodes to control the bandwidth and loss allocations among the individual flows within a CLVL. While OverQoS cannot provide the spectrum of service guarantees offered by IntServ [10], it can still provide useful QoS enhancements to applications. Examples of such enhancements are:

Smoothing losses: Bursty network losses can have a negative impact on many applications such as multi-player games. OverQoS can reduce or even eliminate the loss bursts by smoothing packet losses across time.

Packet prioritization: OverQoS can allow applications to express the importance of the packets within a stream, and protect important packets at the expense of less important ones. For example, OverQoS can protect I-frames in an MPEG stream over B-frames or P-frames.

Statistical Bandwidth and Loss Guarantees: Besides statistical loss guarantees, OverQoS can provide statistical bandwidth guarantees to a small fraction of its traffic.

To understand the tradeoffs and the limitations of the OverQoS architecture, we present its design and implementation, and perform an extensive evaluation. Across a wide-area testbed of 19 diverse nodes (spanning US, Europe, and Asia), we show that OverQoS can simultaneously provide statistical loss guarantees on the order of 0.1% and and bandwidth guarantees ranging from 100 Kbps to 2 Mbps. In addition, by simultaneously running multiple competing CLVLs along with long-lived TCPs on a lossy access network, we show that OverQoS is fair to cross-traffic and can co-exist with other competing OverQoS links.

We additionally demonstrate how *multiplayer games* and *streaming media* can benefit from using OverQoS. In the multi-player game example, an end-user can use OverQoS to interactively play a game like *Counterstrike* in highly lossy environments (experiencing a loss rate as high as 10%) without observing any skips or getting disconnected. In the streaming media example, we demonstrate how *RealPlayer* can use OverQoS to preferentially drop and recover specific packets to enhance the quality of a stream *without consuming any additional bandwidth*. OverQoS achieves this by simply redistributing the losses among the packets within the stream. The increase in the end-to-end delay is bounded by the end-to-end RTT.

The rest of the paper is organized as follows. In Section 2 we describe the basic OverQoS architecture and describe the construction of CLVLs in Section 3. In Section 4, we provide the details of our OverQoS implementation. In Section 5, we show two real-world applications that can benefit by using OverQoS. In Section 6, we evaluate the performance of OverQoS in the wide area Internet. We present related work in Section 7 and conclusions in Section 8.

2 OverQoS Architecture

Figure 1 illustrates an OverQoS network with overlay nodes spanning different routing domains and flows routed within this network. We make no assumptions about the placement of overlay nodes in the Internet. Rather, we assume that a placement of overlay nodes is pre-specified. In this paper, we will assume that the end-to-end path on top of an overlay network is fixed and we will attempt to enhance the QoS along this path in the presence of varying levels of network congestion. We can use existing approaches like RON [32] to determine the overlay path between a pair of end-hosts.

In the remainder of this paper, we will use the term *virtual link* to refer to the IP path between two overlay nodes and *bundle* to refer to a stream of application data packets carried across the virtual link. A bundle typically includes packets from multiple transport-layer flows across different sources and destinations. The following constraints and requirements make the design of any overlay-based QoS challenging:

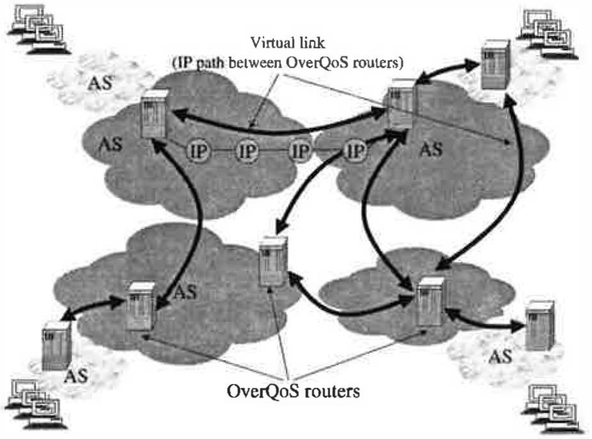


Figure 1: The OverQoS system architecture. OverQoS nodes in different AS's communicate with each other over virtual links using the underlying IP paths.

1. *Node Placement and Cross Traffic:* Overlay nodes will usually span different routing domains and will not be directly connected to the congested links. Hence, one cannot avoid losses or delays along virtual links. Additionally, the losses incurred due to cross traffic is time-varying and can be hard to predict.
2. *Fairness:* Overlays should not offer QoS at the expense of hurting cross traffic. Therefore, the overlay traffic at an aggregate level should be congestion sensitive and not use more than its fair share. One standard metric for determining fair share is based on *TCP-friendliness* [27].
3. *Stability:* Multiple overlay networks independently offering QoS with many virtual links overlapping on congested physical links in the underlying network should be able to co-exist.

To address these challenges we propose a solution that builds on two design principles:

Bundle loss control: Overlay nodes should bound the loss rate experienced by a bundle along a virtual link in the presence of time-varying cross traffic. We propose a *controlled-loss virtual link* (CLVL) abstraction to achieve this loss bound and characterize the service received by a bundle.

Resource management within a bundle: An overlay node can control the loss and bandwidth allocations of each flow and/or application within a bundle.

These design principles enable OverQoS to provide a range of useful services to Internet applications. Example of such services are: (1) packet prioritization, (2) smoothing losses (i.e., eliminate the bursts of losses by spreading losses in time), and (3) statistical bandwidth and loss guarantees, though this service can be typically offered only to a small

b	Maximum sending rate on a virtual link
c	CLVL available/ aggregate bandwidth
q	CLVL target loss rate / statistical bound on the CLVL loss-rate
r	CLVL redundancy factor
c_{min}	Minimum statistical bandwidth guarantee
u	Probability of not meeting the bandwidth guarantee c_{min}

Table 1: OverQoS Notation table

fraction of a bundle's traffic. We next elaborate on our two design principles.

2.1 Bundle Loss Control

The basic building block for enabling OverQoS to achieve loss control over a bundle is the Controlled-loss Virtual Link (CLVL) abstraction. The CLVL abstraction provides a bound, q, on the loss rate seen by the bundle over a certain period of time regardless of how the underlying network loss rate varies with time. Overlays can achieve this bound by recovering from network losses using a combination of Forward Error Correction (FEC) and packet retransmissions in the form of ARQ. By setting q to an arbitrarily low value (close to 0), a CLVL provides the notion of a near-loss free pipe across a virtual link. Therefore, a CLVL *isolates* the losses experienced by the bundle from the loss-rate variations in the underlying IP network path. The biggest challenge in constructing a CLVL is to achieve the loss bound q in the presence of time-varying cross traffic and network conditions. Additionally, the amount of bandwidth overhead should be minimized. In Section 3.2, we present a hybrid FEC/ARQ solution which minimizes the amount of redundancy required to provide a CLVL abstraction for a given value of q.

The total traffic between two overlay nodes consists of: (a) the traffic of the bundle; (b) the redundancy traffic required to achieve the target loss rate, q. The fairness and stability constraints limits the maximum rate (inclusive of the redundancy traffic) at which OverQoS can transmit across a virtual link. Let $b(t)$ denote this traffic bound at time t (Section 3.1 elaborates on how b is computed). Let $r(t)$ denote the fraction of redundancy traffic required by OverQoS to achieve q. Then, the *available bandwidth* for the flows in the bundle is $c(t) = b(t) \times (1 - r(t))$. Thus, the service provided by a CLVL to the bundle is: *As long as the arrival rate of the bundle at the entry node does not exceed $c(t)$, the packet loss rate across the virtual link will not exceed q, with high probability.*

2.2 Resource Management within a Bundle

The CLVL abstraction provides the bundle an available bandwidth, c, which varies with time and guarantees the entire bundle a target loss rate, q. If the traffic arrival rate of the bundle is larger than c, the extra traffic is dropped at the entry overlay node. The overlay node can employ any QoS scheduling discipline to distribute c and the losses across the flows in the bundle. In particular, in a Diffserv-like model, if every packet is associated with a priority, then the overlay node can use these priorities to preferentially drop packets and allocate bandwidth to different flows.

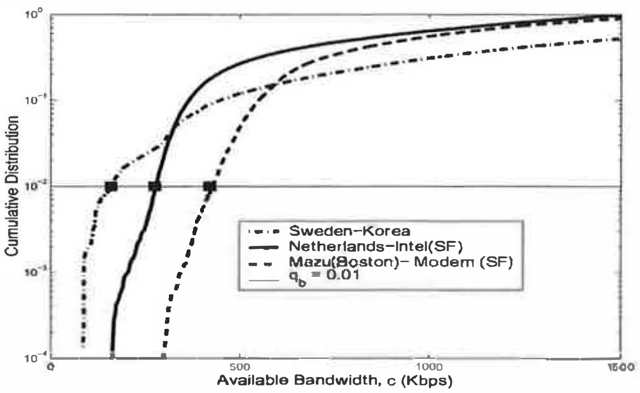


Figure 2: The cumulative distribution of c across three separate CLVLs is measured on Jan 20, 2003 by transmitting 1,500,000 packets over each virtual link(each with 250 bytes payload). The intersection point between $u = 0.01$ and the CDF curves represent the values of c_{min} along the three links.

While in general the available bandwidth, c, of a CLVL bundle varies with time, it might be possible to statistically bound the minimum bandwidth of the bundle to offer bandwidth guarantees to a fraction of OverQoS traffic. Given a small probability value, u, one can capture the variations of the available bandwidth on a CLVL using a distribution and determine a value c_{min} such that the probability, $P(c < c_{min}) = u$ where u represents the probability of not meeting the bandwidth guarantee, c_{min}. If the corresponding c_{min} is a significant fraction of c, then OverQoS can provide statistical bandwidth guarantees by allocating bandwidth to flows within a CLVL as long as the total allocated bandwidth is less than c_{min}. Table 1 tabulates all the variables we use in expressing the properties of a CLVL.

In practice, we notice that the value of c_{min} across overlay links can be reasonably high implying that OverQoS can indeed be used to provide meaningful statistical bandwidth guarantees to applications. Figure 2 shows the distribution of c for three different overlay links traversing international links and broadband networks: Lulea (Sweden)-Korea, Mazu (Boston)- Cable Modem (SF), Netherlands-Intel (SF). The values of c_{min} across these links to provide a $u = 0.01$ guarantee are 160 Kbps, 420 Kbps, and 269 Kbps respectively. Statistical bandwidth guarantees can be provided only to a subset of the OverQoS flows, potentially at the expense of other flows. Flows requiring guarantees should be given a higher priority over other flows at an

OverQoS node. The remaining bandwidth $c - c_{min}$ is distributed among the other flows.

2.3 Overall picture

An OverQoS network (Figure 1) comprises of a collection of overlay links where each link is associated with a CLVL abstraction. Individual CLVLs along different OverQoS links are stitched together to generate an end-to-end path along which a flow may be routed and guaranteed a specific amount of QoS. In this paper, we demonstrate that an overlay network can indeed be useful in enhancing Internet but do not address the issue of how to route flows on top of an OverQoS network. We rely on an overlay routing service like RON [32] to specify an end-to-end path across an OverQoS network. Given one such path, OverQoS determines the level of QoS that can be provided along the path.

Application-OverQoS Interface: A legacy application intending to use OverQoS is required to perform two functionalities. First, it needs to tunnel its packets through the overlay network using an OverQoS proxy. The proxy node functionality can reside either at the first OverQoS node along the path or within the same host as the application. Second, the proxy is responsible for signaling the application specific requirements to OverQoS. For example, if OverQoS offers the service of smoothing losses or packet prioritization, the proxy is required to mark the priority of packets within the flows. Our current implementation of an OverQoS proxy is application specific in that it infers the priorities of the packets of an application flow without modifying the application. However, in the case of statistical loss or bandwidth guarantees, an application is required to clearly signal its QoS requirements (loss,bandwidth) to the OverQoS proxy. For this particular service, the proxy is additionally responsible for undergoing an admission control test to determine whether OverQoS can indeed satisfy the application's QoS requirements. The signaling aspects of the admission control as well as the issue of how to route flows within OverQoS are out of the scope of this paper.

2.4 Discussion

End-to-end Recovery vs Overlay CLVL: An alternative to applying the CLVL abstraction on an overlay network is to apply loss control on an end-to-end per flow basis. There are several arguments against end-to-end loss control: First, using FEC to apply end-to-end loss control is far more expensive than applying it on an aggregate level. For example, in order to provide a 0.1% loss guarantee to a 64 Kbps stream (like game console traffic or IP telephony stream) over a bursty channel with an average loss rate of say 2%, the minimum amount of FEC required can be as high as 32 Kbps. However, if 10 such flows are aggregated at an overlay node, the per-flow FEC requirement can drop to lower than 5 Kbps. Second, with a better distribution of overlay nodes, we expect the overlay links to have much smaller RTTs than end-to-end RTTs. Hence, overlay-level recovery using ARQ has better delay characteristics than end-to-end recovery. Finally, aggregation of flows within an overlay provides the ability to trade resources across different flows (or within packets of the same flow) which is fundamentally necessary to provide better QoS properties.

Delay guarantees: Overlay networks have no control over variations in queuing delays along virtual links and hence cannot offer delay assurances. On the other hand, overlay networks have been used to route around congestion [33, 7]. Such techniques can be embedded into an overlay to improve the end-to-end delay characteristics of a path.

Over-provisioning: Recent measurement studies have shown that Internet backbones are over-provisioned and have low levels of congestion [19, 17]. This questions the basic need for Internet QoS. We contend that over-provisioning is not necessarily a permanent feature of the Internet, but a reflection of the big disparity between the poor connectivity at edges, and the backbone capacity. As more homes and enterprises become connected over faster, multi-megabit/s or higher, links with optical fibers, we expect that at least some parts of the Internet such as small ISPs to become more congested. This trend is already evident in countries like Japan where ISPs offer 10 Mbps broadband connections to homes [4]. In addition, many ISPs already provide aggregate QoS within their networks using MPLS technologies [26]. We believe that overlays are the right platform for translating these aggregate intra-domain QoS to meaningful end-to-end QoS guarantees.

3 Controlled-Loss Virtual Link (CLVL)

In this section, we describe the realization of the CLVL abstraction. In particular, we describe: (a) how to compute, b, the maximum sending rate across an OverQoS link; (b) how to achieve the target loss rate q for the flows in the bundle; (c) the architecture of the OverQoS node.

3.1 Estimating b

OverQoS tunes the maximum output rate, b, depending on network congestion in order to be both fair to cross traffic as well as achieve stability in the presence of other competing OverQoS traffic. One way of achieving this is to set b based on an *N-TCP pipe* abstraction which provides a bandwidth which is N times the throughput of a single TCP connection on the virtual link. We set N to be equal to the number of flows in the bundle.

We use MulTCP [29] to emulate the behavior of N TCP connections. MulTCP uses a TCP-like congestion control mechanism with $\alpha = N/2$ and $\beta = \frac{1}{2N}$ as the increment and decrement parameters. While MulTCP may react quickly to congestion, it may not provide smooth variations in the sending rate. To obtain smoother variations, we may prefer to choose an alternate operating point with a lesser value of α and β without altering the net steady state throughput as determined by the TCP equation [27]. If we set $\alpha = \sqrt{N}$, the corresponding value of β can be calculated using the TCP equation as equal to $4/(3 \times N^{1.5} + 2)$. Across most of our evaluations, we use the standard parameters of MulTCP. Alternatively, we can also use an equation-based approach to emulate the behavior of N TFRC connections [16].

3.2 Achieving target loss rate q

We will describe a hybrid solution which uses a combination of FEC and ARQ to construct a CLVL. Recall that a CLVL abstraction aims to bound the bundle loss rate to a small value q. Since burstiness of cross-traffic is usually unpredictable, we define q as a statistical bound on the average loss rate observed over some larger period of time (on the order of seconds).

FEC vs ARQ trade-off: The main distinction between FEC and ARQ is in the trade-off between bandwidth overhead and packet recovery time. While FEC can help in quickly recovering from packet losses, the bandwidth overhead can be high especially over virtual links experiencing bursty losses [22]. On the other hand, an ARQ based solution will have a high packet recovery time if the RTT between two overlay nodes is large. To strike a balance between these two approaches, we present a hybrid approach that uses the best features of both these mechanisms.

We will first briefly describe how one will construct a CLVL using purely ARQ or FEC and combine these approaches to obtain a hybrid CLVL construction.

ARQ-based CLVL: A purely ARQ-based solution for building CLVLs is easy to construct. In a reliable transmission ($q = 0$), a packet is repeatedly retransmitted until the sender receives an acknowledgment from the receiver. Similarly, to achieve a non-zero target loss rate, q, it is enough to retransmit any lost packet $L = \log_{\bar{p}} q - 1$ times, where $\bar{p}$ represents the average loss rate over the interval over which we want to bound q. However, if $L > 1$, a pure-ARQ based CLVL is unattractive since it uses multiple RTTs to achieve the bound q.

FEC-based CLVL: In an FEC-based approach, we divide time into windows of period, T_r, where a window is a unit of encoding/decoding. We consider an *erasure code* such as Reed-Solomon, characterized by (n, k), where k is the number of packets arriving at the entry node during the window, and $(n - k)$ represents the number of redundant packets added. Let r denote the redundancy factor, where $r = (n - k)/n$. The FEC problem reduces then to determining a minimum redundancy factor, r, such that the target loss rate q is achieved. Since the hybrid approach (*i.e.,* FEC+ARQ based CLVLs) presented below outperforms the FEC based CLVLs in most of the cases, we skip the description of our algorithm for computing the ideal value of r.

FEC+ARQ based CLVL: Due to delay constraints for loss recovery, we restrict the number of retransmissions to at most one. We divide packets into windows and add an FEC redundancy factor of r_1 for each window in the first round. In the second round, if a window is non-recoverable, the entry node retransmits the lost packets with a redundancy factor r_2.

We need to estimate the parameters, r_1 and r_2. Let $f(p)$ denote the PDF of the loss rate p, where each value of p is measured over an encoding/decoding window. FEC offers loss protection within a window if the fraction of packets lost in a window, p, is less than the amount of redundancy added for that window. Given a redundancy factor, r, the expected packet loss rate after recovering from FEC is given by:

$$G(r) = \int_r^1 pf(p)dp$$

Hence the expected packet loss rate after the two rounds in the hybrid approach is equal to $L(r_1, r_2) = G(r_1) \times G(r_2)$. Given a target loss-rate, q, we require:

$$L(r_1, r_2) \leq q$$

For a given window, r_1 is the FEC overhead in the first round, $G(r_1)$ is the expected number of retransmitted packets and $G(r_1) \times r_2$, the expected overhead in the second round. The expected bandwidth overhead is given by

$$O(r_1, r_2) = r_1 + G(r_1)(1 + r_2)$$

This yields the following optimization problem: Given a target loss rate q, determine the redundancy factors r_1 and r_2 that minimizes the expected overhead, $O(r_1, r_2)$ subject to the target loss constraint: $L(r_1, r_2) \leq q$.

For many loss distributions that occur in practice, the optimal solution for this problem is when $r_1 = 0$. This solution implies that it is better not to use FEC in the first round, and use FEC only to protect retransmitted packets. When $r_1 = 0$ and $r_2 = 0$, an FEC+ARQ CLVL reduces to a pure ARQ based CLVL. This happens when $q < p_{avg}^2$ where $p_{avg} = G(0)$ is the average loss-rate along the virtual link. An FEC+ARQ CLVL can be made adaptive to sudden variations in the loss characteristics by always applying a minimal amount of FEC ($r_2 > 0$), to the retransmitted packets in a window.

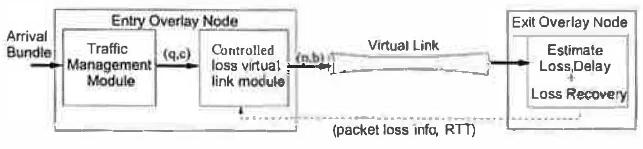

Entry Overlay Node
Arrival Bundle
Traffic Management Module
(q,c)
Controlled loss virtual link module
(n,b)
Virtual Link
Exit Overlay Node
Estimate Loss,Delay + Loss Recovery
(packet loss info, RTT)

Figure 3: Components of entry and exit OverQoS nodes

We made a simplistic assumption in the above calculation. We used the same distribution $f(p)$ to model the fraction of losses during both the first and second round. Since the number of packets in a retransmitted window may be much smaller than the original window, the same distribution $f(p)$ may not apply. To overcome this problem, we estimate a table of loss distributions (rather than one $f(p)$) across different time-scales and apply the appropriate distribution based on the number of retransmitted packets.

3.3 Node Architecture

Figure 3 captures the interactions between the various components in the entry and exit overlay nodes. The entry node consists of two modules: one that implements the CLVL abstraction, and another that performs per-aggregate or per-flow traffic management. The first module communicates with the exit OverQoS node to estimate the link loss rate and delay. It uses this information to adapt the data traffic to conform to the CLVL abstraction. The second module allocates the capacity of the CLVL among competing traffic aggregates or flows. The exit OverQoS node is responsible for measuring the loss and delay characteristics and reconstructing lost packets if necessary. If the CLVL abstraction uses ARQ for loss recovery, the exit node propagates individual packet loss information to the entry node.

The entry node exerts control on the traffic in the bundle at two levels of granularity: on the bundle as a whole, and on a per-flow basis within the bundle. At both these levels, the entry node can control either the sending rate or the loss rate. The CLVL management module at the entry node first determines the sending rate of the bundle, b, using MulTCP [29] to emulate the aggregate behavior of N virtual TCPs. Next, it determines the level of redundancy r required to achieve a certain target loss-rate q based on the loss-characteristics determined by the window. The resulting available bandwidth c is estimated to be $b(1-r)$. The traffic management module at the entry node then distributes the available bandwidth c among the individual flows. If the net input traffic is larger than c, the entry node drops the extra traffic and exercises control in distributing the losses amongst the flows.

4 OverQoS Implementation

We implemented the OverQoS node architecture in about 5000 lines of C code. The communication between overlay nodes uses the UDP socket interface. For loss recovery, we use the FEC software library built by Rizzo *et al.* [30]. Our implementation works on both Linux and FreeBSD platforms.

Figure 4 illustrates the structure of a single OverQoS node along a given path. An OverQoS node listens on a UDP socket for the arrival bundle and tunnels the traffic to the exit node as a UDP stream. The CLVL Encoder and Decoder modules implement the CLVL abstraction on top of the overlay link by adding the necessary level of redundancy to recover from packet losses.The decoder also provides loss feedback to the encoder for computing the optimal redundancy factor. The Traffic Management module implements per-flow or per-packet resource management. Different QoS schedulers and buffer management schemes like priority scheduling and smoothing losses is performed by this module. The rate estimator computes the CLVL parameters b,c and r while the link estimator provides feedback to the transmitting OverQoS node about the virtual link characteristics comprising: (a) loss feedback for computing the loss distribution; (b) RTT, the round trip time.

CLVLs along an overlay path can be stitched together (or *cascaded*) to provide end-to-end services. Cascaded CLVLs can introduce artificial losses at an overlay node if the available bandwidth on the incoming links is larger than the available bandwidth in the outgoing links. In order to avoid any artificial packet losses at an intermediary node in an overlay path, an OverQoS node uses b_{max} to signal the maximum sending rate to its predecessor. This is illustrated in Figure 4.

4.1 Other Implementation Issues

We will now briefly discuss some of the salient implementation issues:

Application-dependent proxy: An important aspect of interfacing with legacy applications is to use an application proxy that can signal an application's requirements to the OverQoS network. In the case of MPEG streaming, the application proxy interprets the packets in the stream and marks the priority of recovery for each packet. For smoothing losses, all packets in a stream are associated with the same priority. For obtaining bandwidth guarantees, the proxy needs to use a signaling mechanism like RSVP [10] to reserve the resources along an overlay path.

Choosing parameters: The parameters N, RTT and p_{avg} need to be estimated for determining the sending rate b. While N can be estimated as the instantaneous number of flows, we set N as the average number of flows observed over a larger period of time (certain flows have a very short lifetime). This is to reduce the variations in the sending rate induced by N. Only flows that generate a minimum

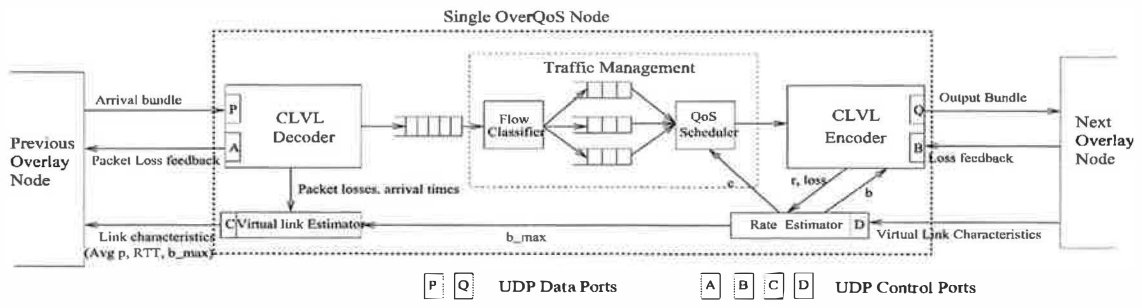


Figure 4: Structure of a single OverQoS node along a path.

number of packets, are used in calculating N. We leverage the techniques used in equation based congestion control [16] for estimating the RTT and p_{avg} between two OverQoS nodes. We choose a reasonably low value of the target loss-rate, $q = 0.1\%$, for most of our experiments. For FEC+ARQ based CLVLs, we choose the packet recovery time, T_r to be $2 \times RTT$.

Startup phase: During periods of no usage (i.e. when N=0), we do not send additional traffic to estimate the virtual link parameters. After such a phase, OverQoS nodes need to determine an initial value of b along a virtual link. Like TCP, we use a *slow-start* phase to estimate the initial value of b. During the slow-start phase, OverQoS does not use loss recovery.

FEC implementation: Our implementation can perform FEC encoding and decoding (for a redundancy factor as high as 50%) at over 300 Mbps on a Pentium III 866 MHz running Linux 2.4.18 kernel. Since we operate on small window sizes, ($n < 1000$), Reed Solomon coding is not a bottleneck. For example, on a virtual link with an $RTT = 100$ ms, the window size is bounded by 1000 for sending rates less than 40 Mbps. Other coding techniques like Tornado codes [23] while faster, may not provide the same level of error correction for small window sizes.

5 Two Sample Applications

In this section, we will describe two real applications that can leverage the QoS enhancements offered by OverQoS. The first application shows how *RealServer*, a streaming media application can improve the signal quality of multimedia streams by using OverQoS to preferentially recover important packets at the expense of less important ones *without using any additional network bandwidth.* The second application is *Counterstrike*, a popular online multiplayer game with a user base of over 1 million players [1]. For this application, we show how OverQoS can smooth out losses and enable players to play the game under high-loss environments.

5.1 Streaming Media Applications

Streaming media applications are typically more sensitive to network losses than delay since delay variations can be masked by using a buffer at the client. OverQoS is an ideal platform for providing different forms of enhancements for such applications. Two such forms of enhancements are:

1. The quality of streaming audio can be enhanced by converting bursty losses into smooth losses.
2. By preferentially recovering packets in an MPEG stream, one can improve the quality of the video stream.

Given that delay variations is not a primary issue for these applications, OverQoS primarily uses an ARQ-based CLVL for these applications. For both streaming audio and video, OverQoS does *not* consume any additional bandwidth. It achieves this by performing the following operation: Whenever an important packet is lost in the network, OverQoS retransmits this packet and drops a later lesser important packet to compensate for the retransmission. In the process, the application observes the same end-to-end loss-rate as it would in the normal Internet and will experience an occasional increase in the end-to-end delay which is bounded by the RTT along the overlay path.

5.1.1 QoS Enhancements for Streaming Media

Streaming Audio: Bursty errors in a streaming audio application can either cause interruptions to an audio stream or cause gaps in an audio stream for periods of time easily perceptible by the human ear. We consider the case where a RealServer streams a *.wav/.mp3* audio file to an end-host using RTP. The audio stream can use OverQoS to smooth out bursty losses *i.e.*, spread a bursty loss over time.

MPEG Streaming: An MPEG video stream consists of a Group of Pictures (GOP) each comprising of I-frames, P-frames and B-frames [6]. Among these, I-frames are the most important since they represent the start of a video sequence in a GOP while P-frames and B-frames are inter-coded frames. Each frame is typically larger than a packet and a frame is sent across multiple consecutive packets. All packets corresponding to an I-frame occur in succession. A single bursty network loss can eliminate an I-frame completely which can cause an MPEG player like Mplayer [5] to disconnect since a GOP cannot be reconstructed. The B-frame and P-frame of a GOP are useless without the corresponding I-frame.

Using OverQoS, one can associate packets belonging to I-frames with higher priority and recover packets within an I-frame at the expense of B or P-frame packets. Additionally, bursty dropping of B and P frame packets affects the quality of one GOP in an MPEG stream, smoothly dropping B and P packets can affect the quality of multiple GOPs. The type of frame of a packet is embedded in the MPEG Video-Specific header within the payload of a packet.

5.1.2 Evaluation

Network Setup: We use the Helix server version 9.0.2 [2] as our streaming media server and use Mplayer [5] as the streaming media client. All streaming media requests are issued using the Real Time Streaming Protocol (RTSP) to stream packets using UDP. We built a client proxy and a server proxy to interpret the streaming media packets and associate them with different priorities. Using these proxies, we tunnel a media stream from RealServer to an Mplayer client along an overlay path along which we replay sample bursty loss traces collected along different overlay links. For the purpose of illustration, we consider two such loss traces: (a) Mazu (Boston)-Korea with an average loss rate of 2%; (b) Intel (San Francisco) - Lulea (Sweden) with an average loss trace of 3%. Each trace is 20 minutes long. To emulate the behavior without OverQoS, we consider the OverQoS nodes to act as packet forwarders. If the length of a media stream is shorter than the length of the trace, we repeat the analysis for different portions of the trace.

Streaming Audio: To demonstrate the effect of smooth dropping on streaming audio, we concatenated several speech samples provided by International Telecommunication Union (ITU-T) to produce two test samples of length 84 sec and 82 sec respectively. Perceptual Evaluation of Speech Quality(PESQ) [3] is one metric to evaluate the quality of voice. We measured the PESQ score for the received stream in comparison to the original stream. A PESQ score of 5 is considered to be ideal implying that the received audio stream has not degraded in quality.[1]

Table 2 compares the PESQ scores of streaming audio with and without OverQoS for two benchmark speech samples. We observe that smoothing the losses does help in increasing the quality of the audio stream. Using OverQoS we are able to increase the PESQ score of the output stream by roughly $0.15 - 0.2$. To demonstrate that $0.15 - 0.2$ is indeed a reasonable improvement in the audio quality, we experimented with several artificial bursty loss patterns while maintaining the same average loss-rate of the traces (*i.e.*, 2% and 3%) and measured the PESQ scores for each of them. For an average loss rate of 2%, we found the PESQ scores to vary between 4.2 and 4.3 across a variety of bursty loss patterns. For these cases, we again found that smooth dropping performs better than bursty drops. Hence, we find that smoothing losses using OverQoS uniformly outperforms different types of bursty network losses.

[1]The PESQ measure is applicable only for pure speech samples and not for arbitrary audio streams. Hence our analysis is limited to only these standardized samples.

		Sample 1	Sample 2
Mazu-Korea	Without OverQoS	4.25 ± 0.3	4.27 ± 0.5
Mazu-Korea	With OverQoS	4.46 ± 0.4	4.45 ± 0.3
Intel-Lulea	Without OverQoS	4.04 ± 0.2	4.13 ± 0.3
Intel-Lulea	With OverQoS	4.19 ± 0.3	4.31 ± 0.3

Table 2: PESQ scores for speech samples with and without OverQoS for both the Mazu-Korea and Intel-Lulea loss traces. This table also shows the standard deviation of these scores across different loss traces.

		5% PSNR	Median PSNR
Mazu-Korea	Without OverQoS	15.27	22.33
Mazu-Korea	Using OverQoS	17.4	24.95
Intel-Lulea	Without OverQoS	14.68	21.59
Intel-Lulea	Using OverQoS	16.21	24.7

Table 3: This table shows the 5% and median values from the PSNR distribution of the received stream. 5% value indicates the minimum PSNR value observed by 95% of the images in the stream.

MPEG streaming: Peak Signal-to-Noise ratio (PSNR) is a standard metric used to measure the quality of the video images in a stream. Given an MPEG stream received at the client, we use the "-yuv4mpeg" utility in Mplayer to convert the stream into a stream of images. For every image, we compute the PSNR value of the received image in comparison to the video image in the original MPEG stream. We quantify the quality of the received MPEG stream using a distribution of PSNR values for the individual images. We consider a sample MPEG-1 stream which is 37 seconds for this analysis.

Table 3 compares the 5% and median values of the PSNR values of the received MPEG stream with and without OverQoS across both the loss samples. We make the following observations. First, in the case when an entire I-frame was lost due to a burst, Mplayer stopped playing the video stream since an entire GOP cannot be reconstructed. This occurred in both the loss traces when a burst coincided with the packets of an I-frame. However, OverQoS was able to recover from the burst so that the stream could progress. Second, OverQoS is able to improve both the 5% and the median PSNR values of the stream by preferentially dropping B and P packets in a burst when compared to the quality of the stream without OverQoS. We illustrate the 5% PSNR value mainly to show that OverQoS not only improves the quality of the stream in the average case but also the minimum quality of a stream. To summarize, OverQoS can improve the quality of a media stream without consuming any additional network resources.

5.2 Counterstrike application

Counterstrike is a team-based multi-player game where online players are grouped into competing teams where each team is assigned a specific goal. The environment of the game is pre-loaded and clients exchange game state over the network using small UDP packets. Bursty losses can have an adverse effect on the progress of this game. First, during the initiation phase, the client generates important control packets which if lost can render the client unable to connect to the server. Second, a burst of packet losses during the middle of a game can either cause skips or cause a player to get disconnected. A skip can arise because the game state messages received immediately after a congestion provides a context jump in the game. Third, in a multiplayer game, problems observed by one player will affect other players in the game. For example, disconnection of a single player can sometimes halt the progress of a game.

OverQoS can alleviate the problem of bursty losses by performing the following operations:

1. Recover from bursty network losses by using an FEC+ARQ based CLVL abstraction between overlay links along the path.
2. Smoothly drop data packets equivalent to the size of the burst at the overlay node.
3. Identify control packets based on packet size and not drop these packets.

By both recovering lost packets as well as smoothly dropping an equivalent amount of data packets at an overlay node, OverQoS achieves three objectives: (a) OverQoS provides the Counterstrike client with critical updates to continue the progress of the game. For example, many UDP packets generated by Counterstrike merely contain the coordinates of different players. If OverQoS can deliver even a fraction of these packets, the client will still be able to reconstruct the movement and position of other players. (b) OverQoS uses minimal amount of additional bandwidth to support this application. The additional bandwidth which is not compensated by OverQoS is the FEC portion of the redundant traffic. Our wide-area experiments over realistic overlay links show that this additional bandwidth is negligible (refer to Section 6). (c) The application observes the same loss-rate as it would in the normal Internet yet not experience any skips in a game. In the event of a bursty loss, the application experiences an additional delay equal to the loss recovery time of a CLVL. With a reasonable distribution of overlay nodes, we expect this recovery time to be much smaller than end-to-end recovery.

Figure 5: Snapshot from a Counterstrike game at a 10% loss rate.

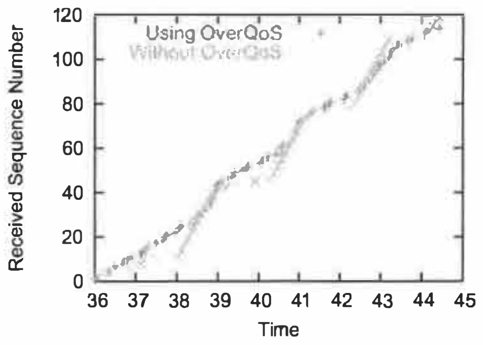


Figure 6: Sequence number plot illustrating smoothing of packet losses using OverQoS.

Counterstrike Proxy: By reverse engineering the traffic characteristics of Counterstrike, we built a client and server proxy to interpret the Counterstrike packets. We chose a proxy-based implementation for two reasons: First, Counterstrike client and server codes are proprietary and we do not modify the code. Second, it is a simple way of capturing different application specific traffic and tunneling them through OverQoS.

Example Scenario: We consider a cable modem loss trace with an high loss-rate of 10% and compare the effect of losses on the Counterstrike game under two scenarios: (a) with OverQoS; (b) without OverQoS. Figure 5 illustrates a snapshot of a Counterstrike game where OverQoS converts bursty losses into smooth losses and the client does not observe any skips. Figure 6 better illustrates the smoothing of losses using OverQoS. In the case without OverQoS, we observed many short periods of time where the network losses was as high as 70 – 80% followed by periods with no congestion. The OverQoS node compensates the additional bandwidth consumed for loss recovery by smoothly dropping packets during non-lossy periods.

We make two additional observations. First, smoothing losses works well only when the bursty loss-periods are relatively short by compensating. When burst periods last for longer periods of time, OverQoS will not be able to smoothly drop packets in the absence of any non-lossy periods. Second, in this scenario, the CLVL abstraction is unable to achieve the target loss-rate due to congestion periods with very high loss-rates. However, the loss reduction provided by OverQoS during bursty periods is sufficient for the Counterstrike game to progress.

6 Evaluation

In this section, we answer several questions relating to the practical viability of OverQoS in the wide area Internet using implementation results and measurements on a wide-area network comprising of 19 diverse nodes. Additionally we use ns-2 based simulations [25] to answer specific questions that a wide area evaluation may not be able to address. The specific questions we address are:

1. Can OverQoS provide statistical bandwidth guarantees and loss assurances to flows? In particular:
   (a) *Loss Guarantees:* When can a CLVL abstraction provide loss guarantees along a virtual link?
   (b) *Bandwidth Guarantees:* What bandwidth guarantees are realizable on a virtual link?
   (c) *OverQoS Cost:* What is the bandwidth overhead and delay cost of using OverQoS?
2. *Fairness/Stability:* Is OverQoS fair to cross traffic and stable in the presence of multiple competing OverQoS networks?

6.1 Evaluation Methodology

Our evaluation methodology is two-fold: (1) we use wide area experiments to evaluate how OverQoS performs in practice, and (2) we use simulations to get a better understanding of the OverQoS performance over a wider range of network conditions.

Wide-Area Evaluation Testbed: Using resources available in two large wide-area test-beds namely RON [32] and PlanetLab [28], we construct a network of 19 nodes in *diverse* locations: 6 university nodes in Europe, 1 site in Korea, 1 in Canada, 3 company nodes, 8 behind access networks (Cable, DSL). Our main goal in choosing these nodes is to test OverQoS across wide-area links which we believe are lossy. For this reason, we avoided nodes at US universities connected to Internet2 which are known to have very few losses [7].

Simulation Environment: We built all the functionalities of our OverQoS architecture on top of the *ns-2* simulator version 2.1b8. Unless otherwise specified, most of our simulations use a simple topology consisting of a single congested link of 10 Mbps where we vary the background traffic to realize different types of traffic loss patterns. We use three commonly used bursty traffic models as background traffic: (a) long lived TCP connections; (b) Self similar traffic [36]; (c) Web traffic [15]. In addition, we use publicly available loss traces to test the performance of a CLVL.

Background Traffic	Average Loss(%)	FEC+ARQ Achieved Loss (%)
100 TCPs(SACK)	1.84	0.06 %
9 Mbps Self Similar	1.91	0.08%
400 Web sessions	0.68	0.03 %

Figure 7: Simulations: Achieved loss rate by a CLVL across three types of background traffic. We set $q = 0.1\%$ and the bottleneck link is 10 Mbps using RED queue.

6.2 Statistical Loss Guarantees

In this section, we answer the question: Under what network conditions, can OverQoS achieve a CLVL abstraction across an overlay link? For all the scenarios described in the section, we choose a target loss-rate to be a small value 0.1%, *i.e.,*, $q = 0.001$. To compute the available bandwidth, b, we use N-TCP with a value of $N = 10$.

Simulations: We first test whether the FEC+ARQ CLVL construction can achieve the target loss-rate across a variety of bursty loss models. Our key conclusion from the simulations is that in *all* cases, we meet the target loss rate $q = 0.1\%$, despite bursty losses and the average loss-rate varying between 0.5% and 3.3%. Furthermore, this conclusion is true not just for the means, but for the tails of the distribution as well. Figure 7 shows the achieved loss rate for the FEC+ARQ based CLVL for three different background traffic scenarios. In addition, our recovery algorithm achieves the target loss irrespective of whether the IP routers along the virtual link use FIFO or RED queues. These results demonstrate that our CLVL algorithm is robust over a range of dynamic traffic conditions and works even when the underlying loss rate is 30 times larger that the target loss rate, q.

Wide Area Evaluation: Given our specific choice of overlay nodes, we found 83 virtual links in our overlay testbed to be lossy. A link is characterized as lossy if the loss-rate along the link is at least 0.5%. Across each link, we ran a CLVL abstraction for time-ranges varying from 20 minutes to 1 hour. In order to measure the system under stress, the sending rate as determined by N-TCP averaged between 120 Kbps (across Cable modems and DSL lines) to 2 Mbps from other nodes. [2]

[2]Given this high sending rate, we did not run our experiments for continued periods of time. Additionally, bandwidth is an ex-

c_{avg}	Mean $\frac{c_{min}}{c_{avg}}$	Variation
100 – 200 Kbps	0.41	0.32 – 0.49
200 – 400 Kbps	0.48	0.29 – 0.75
400 – 800 Kbps	0.41	0.19 – 0.81
800 – 1600 Kbps	0.41	0.16 – 0.86
> 1600 Kbps	0.49	0.04 – 1.0

Figure 9: Variation of $\frac{c_{min}}{c_{avg}}$ as a function of c_{avg}

The FEC+ARQ based CLVL achieved the target loss-rate over 80 of the 83 virtual links. Our FEC+ARQ algorithm failed to achieve the target loss rate of 0.1% only across 3 of the overlay links. Upon closer investigation, we found the causes to be : *short outages* and *bi-modal loss distributions*. A short outage refers to a period of time when all packets transmitted along a virtual link are lost. Within our testbed, we noticed non-recoverable losses along two links: PDI-NBG and Unibo-Media. These non-recoverable losses lasted for short periods of time ($<$ 5 s). Short outages can occur due to a variety of problems such as routing changes or link resets. A loss distribution is said to be bi-modal if the losses experienced in every window is zero or very high. Links with very bursty losses have a bi-modal distribution. An FEC+ARQ based CLVL cannot recover a large portion of a window of packets from a bimodal loss distribution if a long burst affects both the FEC window, and the ARQ transmissions. During our experiments, Mazu-Cba1 experienced a bimodal loss distribution.

6.3 Statistical Bandwidth Guarantees

In this section, we answer the question: What bandwidth guarantees are realizable on a virtual link?

Recall that the statistical bandwidth guarantee achievable along a virtual link is given by c_{min} such that $P(c<c_{min}) = u$, where c represents the instantaneous bandwidth along the virtual link, and u represents the probability with which the guarantee is not met. The Rate Estimator module updates c once every window of packets (O(RTT) sec) based on the feedback information received from the next OverQoS hop.

Across the 171 pairs of nodes between the 19 end-hosts in our testbed, we monitored 83 unique virtual links over a period of 7 working days. Figures 8(a) and (b) show the distribution of c_{min} for $u = 0.01$ and $u = 0.005$. We make two observations. First, the value of c_{min} is greater than 100 Kbps for more than 80% of the links. 20% of the links are predominantly connected to broadband hosts. Second, in many cases, c_{min} is at least 25% of the average throughput along the virtual link. In specific cases, c_{min} is as large as 90% of the average throughput. The median value of

pensive resource for RON and PlanetLab which we did not want to misuse.

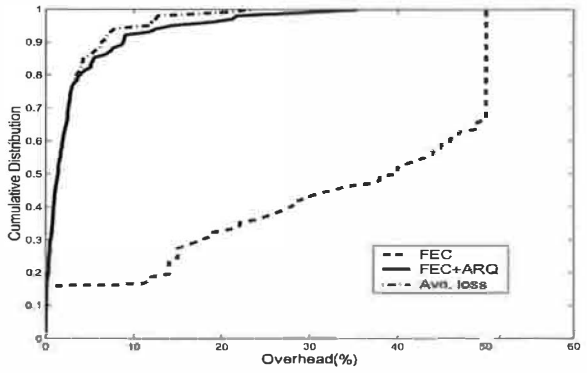


Figure 10: Overhead Characteristics in the wide-area testbed: Compares overhead of FEC+ARQ with FEC and Average loss rate across the links, p_{avg}.

c_{min}/c_{avg} is 0.4 and 0.35 for $u = 0.01$ and $u = 0.005$ respectively. Figure 9 shows the variation of c_{min}/c_{avg} as a function of c_{avg}. As c_{avg} increases, we notice that the maximum value of c_{min}/c_{avg} increases while the minimum value decreases. The minimum decreases because we notice *self-induced losses* across some of the links thereby causing MulTCP to drastically reduce its sending rate and thereby reducing c_{min}.

Stability of c_{min}: If the underlying distribution of c is stable, the estimated value of c_{min} will roughly be a constant. However under dynamic conditions, we need to continuously re-estimate c_{min} and flows need to renegotiate their bandwidth reservations. For a given value of u, we estimate c_{min} using $O(1/u)$ samples of c. As an example, given $RTT = 100$ msec and $u = 0.01$, we can calculate c_{min} based on the last $20/u$ samples (representing a history of 200 seconds). In this scenario, flows renegotiate their bandwidth requirements every few minutes.

Figure 8(c) shows the variation as a function of time across four separate virtual links from Europe to North America. We make two observations: First, the value of c_{min} is very stable compared to variations in the available bandwidth, c. Across these links, c_{min} does not deviate more than 10% around its mean value. Second, an on-line algorithm for estimating c_{min} based on past history is a reasonable approach. While we set $P(c < c_{min})$ to be 1%, the actual value of c is less than the estimated c_{min} in no more than 1.3% of the cases across all four virtual links.

6.4 OverQoS Cost

6.4.1 Overhead Characteristics

Figure 10 shows the cumulative distribution of the overhead for an FEC+ARQ based CLVL across the 83 overlay links over which we performed our measurements. For each link, we ran an $N-$TCP pipe for $N = 10$ and measured the overhead required to achieve a target loss rate of

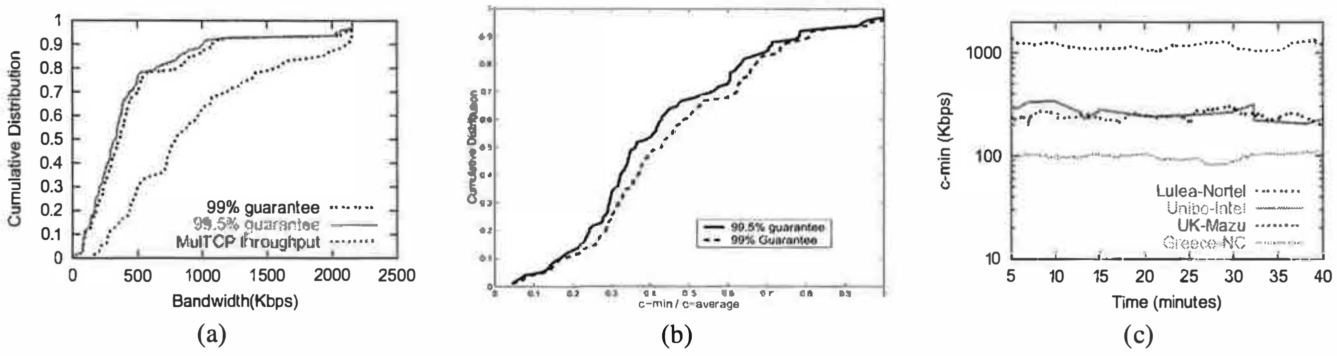


Figure 8: (a) Cumulative distribution of the bandwidth guarantee c_{min} across 100 separate measurements over 83 unique overlay links measured across 7 different days from Jan14th - Jan28th. For each run along a single overlay link, we generated between 100,000 - 300,000 packets. All measurements are taken on weak-days (many of them during working hours). (b) Distribution of the fraction c_{min}/c_{avg} across all the links. (c) Variation of c_{min} across 4 different virtual links between Europe and North America. c_{min} is measured as an on-line estimate over a maximum previous history of 5 minutes (time to collect $20/u$ samples for $u = P(c < c_{min}) = 0.01$).

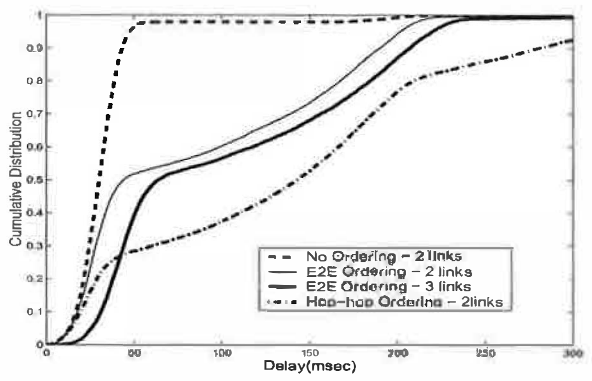


Figure 11: Cascaded CLVL scenario using FEC+ARQ CLVLs: End-to-end ordering within OverQoS network has much better delay characteristics than hop-by-hop ordering.

$q = 0.1\%$. We notice that the overhead of FEC+ARQ is very close to the average loss-rate along the overlay links. The difference between the two is the amount of FEC used in the second round to protect the retransmitted packets. In comparison, a pure FEC based CLVL construction far higher bandwidth. This is primarily due to the network loss characteristics: the burstier the background traffic (i.e., the longer the tail of the loss-rate distribution), the higher the amount of FEC required to recover from these losses [22].

6.4.2 Delay Characteristics

This section answers the question: What is the delay cost of using OverQoS? A potential criticism of our algorithm is that it increases the delay observed by packets.[3] There are two reasons for this increase in delay. First, if one or more packets in a window are lost, the recovery process will cause additional delays. Second, if OverQoS is required to support in-sequence delivery of packets, the loss of one packet can increase the delay of other packets. Our implementation showed that the additional delay incurred at a node due to processing overhead is negligible.

In OverQoS, we can support three different models for packet delivery: (a) No packet ordering; (b) End-to-end (E2E) ordering between first and last OverQoS node in a path; (c) Hop-by-hop ordering. We consider a simple scenario where an overlay path traverses multiple overlay nodes with each link having an RTT of 100 msec and experiencing frequent losses ($p_{avg} = 4\%$). Figure 11 shows the distribution of the additional delay incurred due to loss recovery for each of the three packet delivery models. We consider a path consisting of up to three overlay links. We make three observations. First, end-to-end packet recovery has much better delay characteristics than hop-by-hop delay characteristics. Second, the additional delay incurred by adding new OverQoS nodes along a path is limited. Third, the additional delay is also dependent on the loss rate. The loss-rate dictates how frequently the loss recovery process is being invoked.

6.5 Fairness and Stability

The N-TCP pipe abstraction is built using MulTCP which inherently is TCP-friendly in the aggregate with both cross traffic and other OverQoS traffic. Figure 12 illustrates this fact using a real-world experiment on a link between a university node and NBG, a node behind an access network. Three OverQoS bundles (with N=2, N=4,N=8) compete on this shared bottleneck under two different scenarios: (a) no cross-traffic, and (a) cross-traffic consisting of five long lived TCPs (*wget* downloading content in parallel).

[3]Note that this is a legitimate concern only for OverQoS packets and not for other flows sharing links on a path.

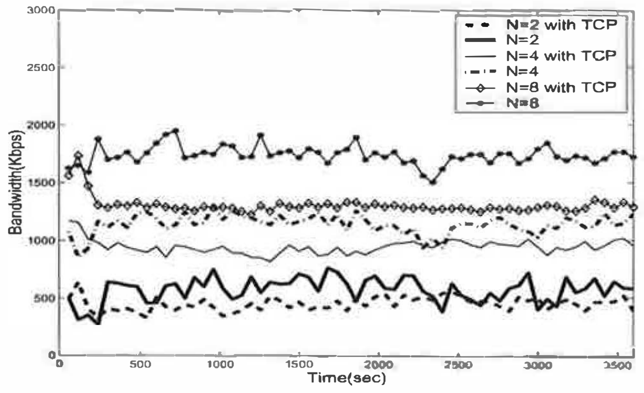


Figure 12: Three independent OverQoS links compete for bandwidth on a shared bottleneck where all CLVLs are established between a university node and NBG, a node behind an access network in Oregon. To make the graph readable, the value of b is averaged over every minute.

We make two observations. First, the three OverQoS bundles co-exist with each other and with the background traffic. Second, the ratio of throughputs of the three OverQoS bundles is preserved across both scenarios.

7 Related Work

We classify related work into: (a) QoS architectures; (b) overlay-based techniques; (c) loss recovery mechanisms.

QoS architectures: OverQoS differs from previously proposed QoS architectures because it does not require QoS mechanisms in all routers in the network. IntServ [10] requires each IP router to implement per-flow admission control on the control path, and per-flow classification, buffer management and scheduling on the data path. Similarly, DiffServ [8, 24] requires edge routers to perform per-flow or per-aggregate classification, buffer management and scheduling, and core routers to perform per-class operations.

OverQoS can leverage the service provided by the underlying network to enhance its services. For instance, within a DiffServ domain, OverQoS may use Expedited Forwarding (or premium service [24]) and provide per-flow bandwidth (and perhaps delay) guarantees. In addition, OverQoS can use techniques like the one proposed in the SCORE architecture [35] to improve its scalability, by having only the first OverQoS node on a flow's path maintain state.

To address the scalability problems of providing end-to-end services, several recent papers have advocated the idea of using endpoint measurement-based admission control (EMBAC) [11, 20, 14]. With EMBAC, an end-host measures the network characteristics of a path and accepts a flow only if the flow's requirements can be satisfied by the path. However, unlike OverQoS, all EMBAC solutions assume that all routers implement some mechanism to isolate the admission-controlled traffic from the best-effort traffic.

Overlay-based Techniques: Several papers have proposed the use of overlay-based approaches for deploying multicast [12, 21] and improving routing functionality (e.g., resilience, as in RON [7]). These systems are motivated in large part by the difficulty of modifying the IP layer both in terms of deployment and in terms of system robustness.

Within the context of QoS, edge-to-edge congestion control [18], a proposal to support a limited range of bandwidth services using an overlay framework, also requires modifications at all edge routers in a domain to achieve its functionality. Service Overlay Network [13], is a recent proposal that purchases bandwidth with certain QoS guarantees from network domains using SLAs and stitches them to provide end-to-end QoS guarantees. Such an architecture would still rely on the underlying domains to meet their specified QoS requirements. For streaming audio and video, multimedia proxies offer the services of smoothing losses [34] and selective discard/recovery of packets [37]. While OverQoS can leverage many of these techniques, two issues differentiate these works from OverQoS: (a) OverQoS can apply the same QoS enhancements within the network as opposed to end-to-end; (b) streaming media flows in OverQoS can be shaped as part of a larger aggregate as opposed to being treated as separate flows.

Loss Recovery: FEC and ARQ based approaches have been investigated in the context of packet audio, video and Internet telephony [9]. Since the FEC constraints are different in these applications (recovering a fraction of packets may be sufficient), we may not be able to apply these results directly to our setting. However, classical coding mechanisms used in wireless networks can potentially be applied to our problem [22, 31, 38].

8 Conclusions

In this paper, we show that it is possible to use overlay networks to enhance Internet QoS without any support from the underlying IP network. Using two real-world applications and experiments over a wide-area testbed we demonstrate three such QoS enhancements: (a) smoothing losses; (b) prioritization of packets within an aggregate; (c) statistical loss and bandwidth guarantees. OverQoS is able to achieve all these enhancements with little (*i.e.,*5%) or no extra bandwidth overhead.

While our results suggest that OverQoS can be a viable architecture to enhance the Internet QoS, more remains to be done. Our current solution assumes that the flows' paths at the OverQoS level are predetermined. A natural extension would be to combine admission control and path selection, e.g., to have the entry OverQoS node compute the "best" path that satisfies a flow's requirements at the admission time. One possibility would be to use RON [32] to find paths with better performance characteristics and to recover from network failures. Another interesting problem would be to determine the "optimal" placement of the OverQoS nodes in the network. We intend to address these issues as part of future work.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Jennifer Rexford, for their invaluable feedback which helped substantially in improving the quality of the paper. We owe our gratitude to Dave Andersen, Tom Anderson, Jon Crowcroft and Scott Shenker for their detailed technical feedback at various stages of this work. We thank Nick Feamster and Matthew Caesar for providing us with MPEG, PESQ processing scripts and several students at UC Berkeley who read earlier drafts of this paper and provided invaluable feedback.

References

[1] CounterStrike. `http://www.counter-strike.net`.

[2] Helix Universal Server - Basic version. `http://www.real.com`.

[3] ITU-T P.862: Perceptual evaluation of speech quality (PESQ). `http://www.itu.int/rec/recommendation.asp?type=items&lang=E&parent=T-REC-P.862-200102-I`.

[4] Japanese broadband statistics. `http://www.johotsusintokei.soumu.go.jp/whitepaper/eng/WP2003/Chapter1-1%.pdf`.

[5] Movie Player for Linux. `http://www.mplayerhq.hu`.

[6] MPEG-1 Specification. `http://www.chiariglione.org/mpeg/standards/mpeg-1/mpeg-1.htm`.

[7] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. ACM SOSP*, Oct. 2001.

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services, Oct. 1998. RFC 2475.

[9] J. Bolot, S. Fosse-Parisis, and D. Towsley. Adaptive FEC-based error control for Internet telephony. In *Proc. of IEEE INFOCOM*, Mar. 1999.

[10] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet architecture: An overview, June 1994. Internet RFC 1633.

[11] L. Breslau, E. W. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint admission control: Architectural issues and performance. In *Proc. of ACM SIGCOMM*, Sept. 2000.

[12] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the Internet using an overlay multicast architecture. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[13] Z. Duan, Z. Zhang, and Y.T.Hou. Service Overlay Networks: SLA, QoS and bandwidth provisioning. In *Proc. of ICNP*, Nov. 2002.

[14] V. Elek, G. Karlsson, and R. Ronngren. Admission control based on end-to-end measurements. In *Proc. of IEEE INFOCOM*, Mar. 2000.

[15] A. Feldmann, P. Huang, A. C. Gilbert, and W. Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proc. of ACM SIGCOMM*, Aug. 1999.

[16] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. of ACM SIGCOMM*, Aug. 2000.

[17] C. Fraleigh, F. Tobagi, and C. Diot. Provisioning IP backbone networks to support latency sensitive traffic. In *Proc. of IEEE INFOCOM*, Mar. 2003.

[18] D. Harrison, S. Kalyanaraman, and S. Ramakrishnan. Overlay Bandwidth Services: Basic framework and an edge-to-edge closed-loop building block, Jan. 2001. Preprint.

[19] `http://ipmon.strintlabs.com`.

[20] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for Integrated Services packet networks. In *Proc. of SIGCOMM*, 1995.

[21] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. USENIX OSDI*, Oct. 2000.

[22] S. Lin and D. Costello. Error Control coding: Fundamentals and applications. In *Prentice Hall*, Feb. 1983.

[23] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss resilient codes. In *Proc. of ACM STOC*, 1998.

[24] K. Nichols, V. Jacobson, and L. Zhang. An approach to service allocation in the Internet, Nov. 1997. Internet Draft.

[25] Ucb/lbnl/vint network simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/.

[26] E. Osborne and A. Simha. Traffic Engineering with MPLS. In *Cisco Press*, July 2002.

[27] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of ACM SIGCOMM*, Oct. 1998.

[28] `http://www.planet-lab.org`.

[29] P.Oechslin and J.Crowcroft. Weighted proportionally fair differentiated service tcp. In *Proc. of ACM Computer Communications Review*, 1998.

[30] L. Rizzo. `http://info.iet.unipi.it/~luigi/fec.html`.

[31] L. Rizzo and L. Vicisano. RMDP: An FEC-based reliable multicast protocol for wireless environments. *Mobile Computing and Communications Review*, 2(2), 1998.

[32] Resilient Overlay Networks. `http://nms.lcs.mit.edu/ron/`, 2001.

[33] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *Proc. of ACM SIGCOMM*, Aug. 1999.

[34] S. Sen, J. Rexford, J. Dey, J. Kurose, and D.Towsley. Online smoothing of variable-bit-rate streaming video. In *IEEE Trans. on Multimedia*, Mar. 2000.

[35] I. Stoica and H. Zhang. Providing Guaranteed Services without per flow management. In *Proc. of ACM SIGCOMM*, Sept. 1999.

[36] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. In *Proc. of ACM SIGCOMM*, Aug. 1995.

[37] Z. Zhang, S. Nelakuditi, R. Aggarwal, and R. Tsang. Efficient selective frame discard algorithms for stored video delivery across resource constrained networks. In *Proc. of IEEE INFOCOM*, 1999.

[38] M. Zorzi. Performance of FEC and ARQ in bursty channels under delay constraints. In *Proc. of VTC'98*, May 1998.

Designing a DHT for low latency and high throughput

Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, Robert Morris *
MIT Computer Science and Artificial Intelligence Laboratory
fdabek, jinyang, sit, jsr, kaashoek, rtm@csail.mit.edu

Abstract

Designing a wide-area distributed hash table (DHT) that provides high-throughput and low-latency network storage is a challenge. Existing systems have explored a range of solutions, including iterative routing, recursive routing, proximity routing and neighbor selection, erasure coding, replication, and server selection.

This paper explores the design of these techniques and their interaction in a complete system, drawing on the measured performance of a new DHT implementation and results from a simulator with an accurate Internet latency model. New techniques that resulted from this exploration include use of latency predictions based on synthetic coordinates, efficient integration of lookup routing and data fetching, and a congestion control mechanism suitable for fetching data striped over large numbers of servers.

Measurements with 425 server instances running on 150 PlanetLab and RON hosts show that the latency optimizations reduce the time required to locate and fetch data by a factor of two. The throughput optimizations result in a sustainable bulk read throughput related to the number of DHT hosts times the capacity of the slowest access link; with 150 selected PlanetLab hosts, the peak aggregate throughput over multiple clients is 12.8 megabytes per second.

1 Introduction

The Internet has transformed communication for distributed applications: each new system need not implement its own network, but can simply assume a shared global communication infrastructure. A similar transformation might be possible for storage, allowing distributed applications to assume a shared global storage infrastructure. Such an infrastructure would have to name and find data, assure high availability, balance load across available servers, and move data with high throughput and low latency.

Distributed hash tables (DHTs) are a promising path towards a global storage infrastructure, and have been used as the basis for a variety of wide-area file and content publishing systems [13, 26, 34, 38]. Good performance, however, is a challenge: the DHT nodes holding the data may be far away in the network, may have access link capacities that vary by orders of magnitude, and may experience varying degrees of congestion and packet loss.

This paper explores design choices for DHT read and write algorithms. Existing work has investigated how to make the *lookup of keys* in DHTs scalable, low-latency, fault-tolerant, and secure, but less attention has been paid to the efficiency and robustness with which DHTs *read and store data*. This paper considers a range of design options for efficient data handling in the context of a single DHT, DHash++. The decisions are evaluated in simulation and in an implementation of DHash++ on the PlanetLab [29] and RON [2] test-beds.

To bound the discussion of design decisions, we have made a number of assumptions. First, we assume that all nodes cooperate; the algorithms for reading and writing are likely to be more expensive if they have to defend against malicious nodes. Second, we assume that lookups are routed using one of the $O(\log N)$-style schemes, instead of using the recently proposed $O(1)$ schemes [14, 17, 18, 44]. Finally, we assume that the DHT stores small blocks (on the order of 8192 bytes). Relaxing these assumptions will result in different DHT designs with different latency and throughput properties, which we hope to explore in the future.

The paper makes the following contributions. Recursive lookups take about 0.6 times as long as iterative; the reason why the reduction is not a factor of two is the cost of the final return trip. The latency of the last few hops in a lookup acts as a lower bound on the performance of Proximity Neighbor Selection [37, 16], which approximates 1.5 times the average round trip time in the underlying network. This result holds regardless of the number of DHT nodes (and thus regardless of the number of hops). Replicated data allows for low-latency reads because there are many choices for server selection, while erasure-coded data reduces bandwidth consumption for writes at the expense of increased read latency. Integration of key lookup and data fetch reduces the lower bound imposed by the last few lookup hops. Finally, using an integrated trans-

*This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

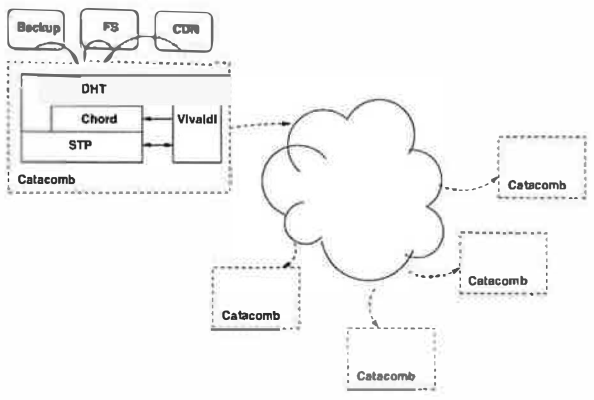


Figure 1: DHash++ system overview.

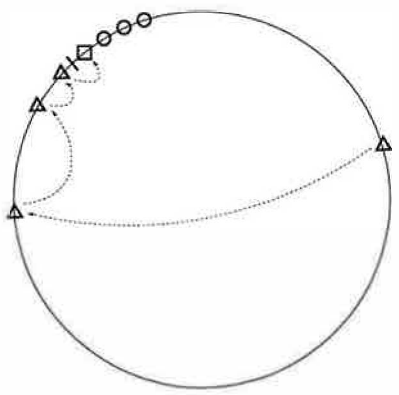


Figure 2: An illustration of a Chord identifier ring. The tick mark denotes the position of a key in ID space. The square shows the key's successor node, and the circles show the nodes in the successor's successor list. The triangles and arrows show a lookup path. The last node before the tick mark is the key's predecessor.

port protocol rather than TCP provides opportunities for efficiency in alternate routing after timeouts and allows the DHT freedom to efficiently contact many nodes.

The rest of this paper is structured as follows. Section 2 outlines the complete system that surrounds the specific mechanisms detailed in the paper. Section 3 describes the methods behind the paper's measurements and quantitative evaluations. Section 4 discusses design decisions that affect latency, and Section 5 discusses throughput. Section 6 describes related work. We conclude in Section 7.

2 Background

For concreteness, this evaluates design decisions in the context of a complete DHT called DHash++. This section describes the parts of DHash++ that are needed to understand the rest of the paper.

2.1 Chord

DHash++ uses the Chord lookup algorithm to help it find data [42]. Chord provides a function `lookup(key)` $\rightarrow$ `set-of-IP`, which maps a 160-bit key to the set of IP addresses of the nodes responsible for that key. Each node has a 160-bit identifier, and Chord designates the s nodes whose identifiers immediately follow a key as responsible for that key; these are the key's *successors*. To provide reliable lookup even if half of the nodes fail in a 2^{16}-node network, the number of successors, s, is 16 in the Chord implementation. The ID space wraps around, so that zero immediately follows $2^{160} - 1$.

The *base Chord* lookup algorithm (which will be modified in subsequent sections) works as follows. Each Chord node maintains a *finger table*, consisting of the IP addresses and IDs of nodes that follow it at power-of-two distances in the identifier space. Each node also maintains a *successor list* referring to its s immediate successors. When a node originates a lookup, it consults a sequence of other nodes, asking each in turn which node to talk to next. Each node in this sequence answers with the node from its finger table with highest ID still less than the desired key. The originating node will find the key's *predecessor* node after $O(\log N)$ consultations; it then asks the predecessor for its successor list, which is the result of the lookup. This style of lookup is called *iterative*, since the originating node controls each step of the lookup. All of the communication uses UDP RPCs.

Figure 2 shows a Chord ring with a key, its successor, the successor's successor list, and a lookup path; this picture is helpful to keep in mind since much of the discussion appeals to the ring geometry. Although this paper explores optimizations over base Chord, we believe that these optimizations also apply to other DHTs that route in ID spaces using an $O(\log N)$ protocol.

2.2 DHash++

DHash++ stores key/value pairs (called blocks) on a set of servers. The DHash++ client API consists of `key` $\leftarrow$ `put(value)` and `get(key)` $\rightarrow$ `value`. DHash++ calculates the key to be the SHA-1 hash of the value, and uses Chord to decide which server should store a given block; each server runs both Chord and DHash++ software. As well as finding and moving data for client applications, DHash++ authenticates the data and moves it from server to server as nodes join, leave, and fail [7].

2.3 Synthetic coordinates

Many of the techniques described in this paper use synthetic coordinates to predict inter-node latencies without having to perform an explicit measurement to determine the latency. A number of synthetic coordinate systems have been proposed [10, 24, 27, 30, 33, 39]. We chose to use Vivaldi [12], because its algorithm is decentralized, which makes it suitable for use in peer-to-peer systems.

Furthermore, the Vivaldi algorithm is lightweight, since it can piggy-back on DHash++'s communication patterns to compute coordinates.

Whenever one Chord or DHash++ node communicates directly with another, they exchange Vivaldi coordinates. Nodes store these coordinates along with IP addresses in routing tables and successor lists. The result of a lookup for a key carries the coordinates of the nodes responsible for the key as well as their IP addresses. Thus the requesting node can predict the latencies to each of the responsible nodes without having to first communicate with them.

3 Evaluation methods

The results in this paper are obtained through simulations and measurements on the PlanetLab and RON test-beds. The measurements focus on DHT operations that require low latency or high throughput.

3.1 Evaluation infrastructure

DHT performance depends on the detailed behavior of the servers and the underlying network. The test-bed measurements in Section 4 were taken from a DHash++ implementation deployed on the PlanetLab and RON test-beds. 180 test-bed hosts were used, of which 150 were in the United States and 30 elsewhere. 105 of the hosts are on the Internet2 network; the rest have connections via DSL, cable modem, commercial T1 service, or are at co-location centers. Each host runs three independent DHash++ processes, or *virtual nodes*, in order to improve load balance and to ensure that the total number of nodes is large compared to the size of the Chord successor list. The measurements in Section 5 were taken on the 27-node RON test-bed alone.

The test-bed measurements are augmented with simulation results to explore large configurations, to allow easy testing of alternate designs, and to allow analytic explanations of behavior in a controlled environment. The simulated network models only packet delay. One input to the simulator is a full matrix of the round-trip delays between each pair of simulated hosts. This approach avoids having to simulate the Internet's topology, a currently open area of research; it requires only the measurement of actual pair-wise delays among a set of hosts. The simulator can produce useful speed-of-light delay results, but cannot be used to predict throughput or queuing delay.

The simulator's delay matrix is derived from Internet measurements using techniques similar to those described by Gummadi et al. [15]. The measurements involved 2048 DNS servers found with inverse DNS lookups on a trace of over 20,000 Gnutella clients. For each pair of these servers, a measuring node sends a query to one server that requires it to contact the other server. Subtracting the delay between the measuring node and the first server from

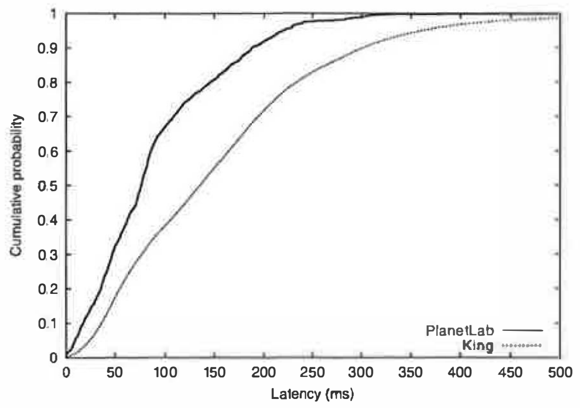


Figure 3: Round-trip latency distribution over all pairs of PlanetLab and King dataset hosts. The median and average King dataset latencies are 134 and 154 milliseconds respectively. The median and average PlanetLab latencies are 76 and 90 milliseconds respectively.

the total delay yields the delay between the two servers. In order to reduce the effects of queuing delay, the minimum delay from five experiments is used. In this paper the results are called the King data-set. All the simulations in this paper involve 2048 DHT nodes using King delay matrix unless otherwise mentioned. Figure 3 shows the CDF of the King data-set round-trip times; the median is 134 milliseconds, while the average is 154 milliseconds. The graph also shows the minimum delay of five pings between each pair of PlanetLab hosts for comparison. The main difference between the two curves is the longer tail on the King distribution, which is likely caused by the larger sample of nodes.

3.2 Application workload

The design of a DHT must incorporate assumptions about probable application behavior, and a DHT evaluation must also involve either applications or models of application behavior. The application aspects that most affect performance are the mix of read and write operations, the degree to which operations can be pipelined, and the size of the data records.

DHash++ is designed to support read-heavy applications that demand low-latency and high-throughput reads as well as reasonably high-throughput writes. Examples of such applications might include the Semantic Free Referencing system (SFR) [45] and UsenetDHT [40].

SFR is a naming system designed to replace the use of DNS as a content location system. SFR uses a DHT to store small data records representing name bindings. Reads are frequent and should complete with low latency. Writes are relatively infrequent and thus need not be as high performance. SFR data blocks are likely to be on the order hundreds of bytes.

UsenetDHT is a service aiming to reduce the total storage dedicated to Usenet by storing all Usenet articles in a shared DHT. UsenetDHT splits large binary articles (averaging 100 KB) into small blocks for load balance, but smaller text articles (typically 5 KB or less) are stored as single blocks. While readership patterns vary, UsenetDHT must support low-latency single article reads, as well as high-throughput pipelined article fetches.

These systems are unlikely to be deployed on high-churn networks—these systems are all server-class. The target environment for them is a network with relatively reliable nodes that have good Internet access.

4 Designing for low latency

This section investigates five design choices that affect DHT `get` latency. The naive algorithm against which these choices are judged, called *base DHash++*, operates as follows. Each 8192-byte block is stored as 14 1171-byte erasure-coded fragments, any seven of which are sufficient to reconstruct the block, using the IDA coding algorithm [31]. The 14 fragments are stored at the 14 immediate successors of the block's key. When an application calls `get(key)`, the originating node performs an iterative Chord lookup, which ends when the key's predecessor node returns the key's 16 successors; the originating node then sends seven parallel requests the first seven successors asking them each to return one fragment.

Figure 4 gives a preview of the results of this section. Each pair of bars shows the median time to fetch a block on the PlanetLab test-bed after cumulatively applying each design improvement. The design improvements shown are recursive rather than iterative routing, proximity neighbor selection, fetching of data from the closest copy, and integration of lookup routing and data fetching. These design improvements together reduce the total fetch latency by nearly a factor of two.

This paper uses a $log(N)$ protocol for routing lookups. An optimization that isn't explored in this paper is an increase in the base to reduce the number of hops, or the use of a constant-hop protocols. These optimizations would reduce latency under low churn, because each node would know about many other nodes. On the other hand, in high churn networks, these optimizations might require more bandwidth to keep routing tables up to date or experience more timeouts because routing tables might contain recently-failed nodes. The paper's evaluation infrastructure isn't adequate to explore this design decision in detail. We hope to explore this issue in future work. We do explore the extent to which proximity routing can reduce the impact of the number of hops on the lookup latency.

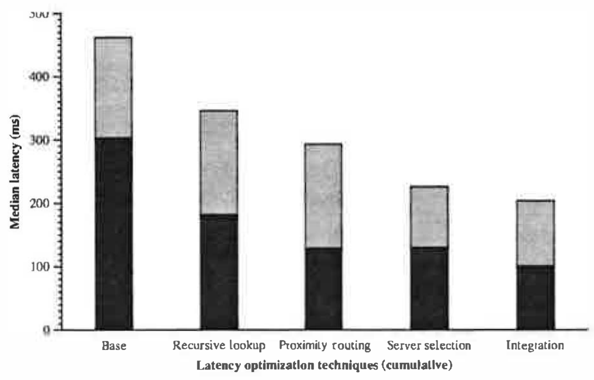


Figure 4: The cumulative effect of successive optimizations on the latency of a DHash++ data fetch. Each bar shows the median time of 1,000 fetches of a randomly chosen 8192-byte data block from a randomly chosen host. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data. These data are from the implementation running on PlanetLab.

4.1 Data layout

The first decision to be made about where a DHT should store data is whether it should store data at all. A number of DHTs provide only a key location service, perhaps with a layer of indirection, and let each application decide where (or even whether) to store data [20, 28]. The choice is a question of appropriate functionality rather than performance, though Section 4.5 describes some performance benefits of integrating the DHT lookup and data storage functions. The approach taken by DHash++ is appropriate for applications that wish to view the DHT as a network storage system, such as our motivating examples SFR and UsenetDHT.

For DHTs that store data, a second layout decision is the size of the units of data to store. A DHT key could refer to a disk-sector-like block of data [13], to a complete file [38], or to an entire file system image [11]. Large values reduce the amortized cost of each DHT lookup. Small blocks spread the load of serving popular large files. For these reasons, and because some applications such as SFR require the DHT to store small blocks, DHash++ is optimized with blocks of 8 KB or less in mind.

A third layout decision is which server should store each block of data (or each replica or coded fragment). If a given block is likely to be read mostly by hosts in a particular geographic area, then it would make sense to store the data on DHT servers in that area. Caching is one way to achieve this kind of layout. On the other hand, geographic concentration may make the data more vulnerable to network and power failures, it may cause the load to be less evenly balanced across all nodes, and is difficult to arrange in general without application hints. At the other extreme, the DHT could distribute data uniformly at random over the available servers; this design would be reasonable if there were no predictable geographic locality in the originators of requests for the data, or if fault-tolerance were important. DHash++ uses the latter approach: a block's key is essentially random (the SHA-1 of the block's value), node IDs are random, and a block's replicas or fragments are placed at its key's successor nodes. The result is that blocks (and load) are uniformly spread over the DHT nodes, and that a block's replicas or fragments are widely scattered to avoid correlated failure.

Given a DHT design that stores blocks on randomly chosen servers, one can begin to form some expectations about fetch latency. The lower bound on the total time to find and fetch a block is the round trip time from the originator to the nearest replica of the block, or the time to the most distant of the closest set of fragments required to reconstruct the block. For the typical block this time is determined by the distribution of inter-host delays in the Internet, and by the number of choices of replicas or fragments. The DHT lookup required to find the replicas or fragments will add to this lower bound, as will mistakes in predicting which replica or fragments are closest.

Most of the design choices described in subsequent subsections have to do with taking intelligent advantage of choices in order to reduce lookup and data fetch latency.

4.2 Recursive or iterative?

The base Chord and Kademlia algorithms are iterative: the originator sends an RPC to each successive node in the lookup path, and waits for the response before proceeding [25, 42]. Another possibility is recursive lookup [6, 47]: each node in the lookup path directly forwards the query to the next node, and when the query reaches the key's predecessor, the predecessor sends its successor list directly back to the originator [42]. Recursive lookup, which many DHTs use, might eliminate half the latency of each hop since each intermediate node can immediately forward the lookup before acknowledging the previous hop.

Figure 5 shows the effect of using recursive rather than iterative lookup in the simulator with the 2048-node King data set. For each technique, 20,000 lookups were performed, each from a random host for a random key. The average number of hops is 6.3. Recursive lookup takes on average 0.6 times as long as iterative. This decrease is not quite the expected factor of two: the difference is due to the extra one-way hop of (on average) 77 milliseconds to return the result to the originator.

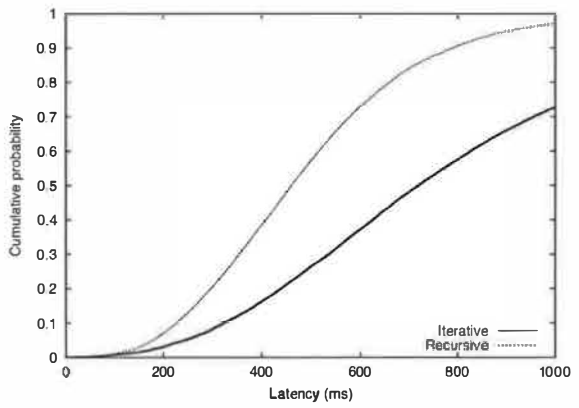


Figure 5: The cumulative distributions of lookup time for Chord with recursive and iterative lookup. The recursive median and average are 461 and 489 milliseconds; the iterative median and average are 720 and 822 milliseconds. The numbers are from simulations.

While recursive lookup has lower latency than iterative, iterative is much easier for a client to manage. If a recursive lookup elicits no response, the originator has no information about what went wrong and how to re-try in a way that is more likely to succeed. Sometimes a simple re-try may work, as in the case of lost packets. If the problem is that each successive node can talk to the next node, but that Internet routing anomalies prevent the last node from replying to the originator, then re-tries won't work because only the originator realizes a problem exists. In contrast, the originator knows which hop of an iterative lookup failed to respond, and can re-try that hop through a different node in the same region of the identifier space.

On the the other hand, recursive communication may make congestion control easier (that is, it is it may make it more feasible to rely on TCP). We will show in Section 5 that the performance of a naive TCP transport can be quite poor.

DHash++ uses recursive lookups by default since they are faster, but falls back on iterative lookups after persistent failures.

4.3 Proximity neighbor selection

Many DHTs decrease lookup latency by choosing nearby nodes as routing table entries [6, 16, 25, 42, 43, 47], a technique often called proximity neighbor selection (PNS). The reason this is possible is that there are usually few constraints in the choice of routing entries: any node in the relevant portion of the identifier space is eligible. A DHT design must include an algorithm to search for nearby nodes; an exhaustive search may improve lookup latency, but also consume network resources. This sub-

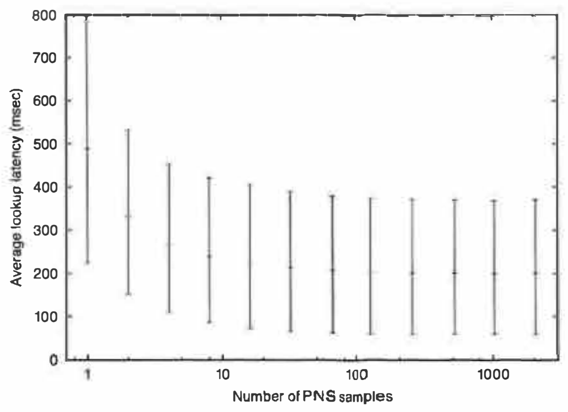


Figure 6: Average lookup latency as a function of the number of PNS samples. The bar at each x value shows the 10th, average, and 90th percentile of the latencies observed by 20,000 recursive lookups of random keys from random nodes using PNS(x). The measurements are from the simulator with 2048 nodes.

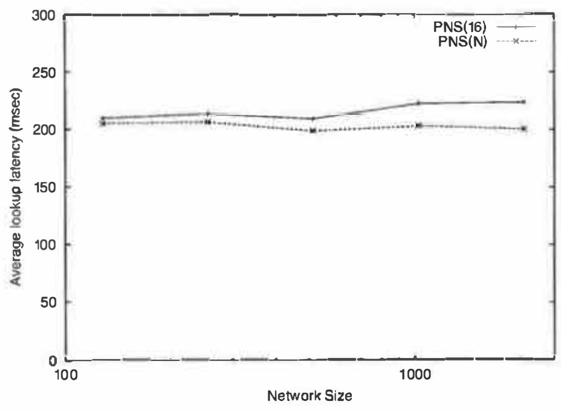


Figure 7: Average lookup latency of PNS(16) and PNS(N) as a function of the number of nodes in the system, N. The simulated network sizes consist of 128, 256, 512, 1024, 2048 nodes.

section builds on the work of Gummadi et al. [16] in two ways: it explains why PNS approximates 1.5 times the average round trip time in the underlying network and shows that this result holds regardless of the number of DHT nodes (and thus regardless of the number of hops).

Following Gummadi et al. [16], define PNS(x) as follows. The ith Chord finger table entry of the node with ID a properly refers to the first node in the ID-space range $a + 2^i$ to $a + 2^{i+1} - 1$. The PNS(x) algorithm considers up to the first x nodes in that range (there may be fewer than x), and routes lookups through the node with lowest latency. *Ideal PNS* refers to PNS(x) with x equal to the total number of nodes, so that every finger table entry points to the lowest-latency node in the entire allowed ID-space range. The simulator simply chooses the lowest-latency of the x nodes, while the real implementation asks each proper finger entry for its successor list and uses Vivaldi to select the closest node. This means that the real implementation requires that $x \leq s$ (the number of successors).

What is a suitable value for x in PNS(x)? Figure 6 shows the simulated effect of varying x on lookup latency. For each x value, 20,000 lookups were issued by randomly selected hosts for random keys. Each lookup is recursive, goes to the key's predecessor node (but not successor), and then directly back to the originator. The graph plots the median, 10th percentile, and 90th percentile of latency.

Figure 6 shows that PNS(1) has a simulated average latency of 489 ms, PNS(16) has an average latency of 224 ms, and PNS(2048) has an average latency of 201 ms. The latter is ideal PNS, since the neighbor choice is over all nodes in the simulation. PNS(16) comes relatively close to the ideal, and is convenient to implement in the real system with successor lists.

Why does ideal PNS show the particular improvement that it does? The return trip from the predecessor to the originator has the same median as the one-way delay distribution of the nodes in the network, δ. For the King data set, $\delta = 67$ms. The last hop (to the predecessor) has only one candidate, so its median latency is also δ. Each preceding hop has twice as many candidate nodes to choose from on average, since the finger-table interval involved is twice as large in ID space. So the second-to-last hop is the smaller of two randomly chosen latencies, the third-to-last is the smallest of four, etc. The minimum of x samples has its median at the $1 - 0.5^{\frac{1}{x}}$ percentile of the original distribution, which can be approximated as the $\frac{1}{x}$ percentile for large x. Doubling the sample size x will halve the percentile of the best sample. Assuming a uniform latency distribution, doubling the sample size halves the best sampled latency. Therefore, the latencies incurred at successive lookup hops with ideal PNS can be approximated by a geometric series with the final lookup hop to the key's predecessor being the longest hop. The lookup process includes an additional final hop to the originator. If we use the per-hop median latency as a gross approximation of the average per-hop latency, the total average lookup latency is thus approximated as: $\delta + (\delta + \frac{\delta}{2} + \frac{\delta}{4} + ...) = \delta + 2\delta = 3\delta$. For the King data set, this gives 201 ms. This is coincidentally the ideal PNS simulation result of 201 ms.

The fact that the average lookup latency of PNS(N) can be approximated as an infinite geometric series whose sum converges quickly suggests that despite the fact that

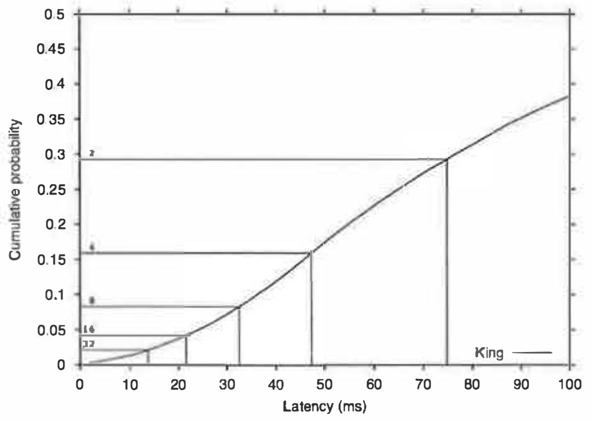


Figure 8: The median of the minimum latency taken from x samples out of the all-pairs empirical latency distribution of the King dataset. The boxes correspond to 2,4,8,16,32 samples starting from the right.

the number of lookup hops scales as $log(N)$, the total average lookup latency will stay close to 3δ. Figure 7 shows the simulated average lookup latency as a function of the number of nodes in the system. As we can see, there is indeed little increase in average lookup latency as the network grows.

Why are there diminishing returns in Figure 6 beyond roughly PNS(16)? First, the King delay distribution is not uniform, but has a flat toe. Thus increasing the number of samples produces smaller and smaller decreases in minimum latency. Figure 8 shows this effect for various sample sizes. Second, for large x, the number of samples is often limited by the allowed ID-space range for the finger in question, rather than by x; this effect is more important in the later hops of a lookup.

One lesson from this analysis is that the last few hops of a lookup dominate the total latency. As a lookup gets close to the target key in ID space, the number of remaining nodes that are closer in ID space to the key decreases, and thus the latency to the nearest one increases on average. Section 4.5 shows how to avoid this problem.

4.4 Coding versus replication

Once the node originating a fetch acquires the key's predecessor's successor list, it knows which nodes hold the block's replicas [13, 38] or fragments of an erasure-coded block [8, 3, 22, 19]. In the case of replication, the originator's strategy should be to fetch the required data from the successor with lowest latency. The originator has more options in the case of coded fragments, but a reasonable approach is to fetch the minimum required number of fragments from the closest successors. The technique of fetching the data from the nearest of a set of candidate nodes is typically called server selection.

The design choice here can be framed as choosing the coding parameters l and m, where l is the total number of fragments stored on successors and m is the number required to reconstruct the block. Replication is the special case in which $m = 1$, and l is the number of replicas. The *rate* of coding, $r = \frac{l}{m}$, expresses the amount of redundancy. A replication scheme with three replicas has $m = 1$, $l = 3$, and $r = 3$, while a 7-out-of-14 IDA coding scheme has $m = 7$, $l = 14$, and $r = 2$.

The choice of parameters m and l has three main effects. First, it determines a block's availability when nodes fail [46]. If the probability that any given DHT node is available is p_0, the probability that a block is still available is [4]:

$$p_{avail} = \sum_{i=m}^{l} \binom{l}{i} p_0^i (1 - p_0)^{l-i} \quad (1)$$

Second, increasing r is likely to decrease fetch latency, since that provides the originator more choices from which to pick a nearby node. Third, increasing r increases the amount of communication required to write a block to the DHT. These performance aspects of erasure coding have not been considered previously.

Figure 9 illustrates the relationship between total fetch latency and block availability. The probability p_0 that each node is available is kept constant at 0.9. Each line represents a different rate r, and the points on the line are obtained by varying m and setting $l = r \times m$. Each point's x-axis value indicates the probability that a block is available as calculated by Equation 1. Each point's y-axis value is the average latency from 20,000 simulations of fetching a random block from a random originating node. The originator performs a lookup to obtain the list of the desired key's successors, then issues parallel RPCs to the m of those successors that have lowest latency, and waits for the last of the RPCs to complete. The y-axis values include only the data fetch time.

The left-most point on each line corresponds to replication; that point on the different lines corresponds to 2, 3, and 4 replicas. For each line, the points farther to the right indicate coding schemes in which smaller-sized fragments are placed onto larger numbers of nodes. For each redundancy rate r, replication provides the lowest latency by a small margin. The reason is easiest to see for $r = 2$: choosing the nearest k of $2k$ fragments approaches the median as k grows, while choosing the nearest replica of two yields a latency considerably below the median. Replication also provides the least availability because the redundant information is spread over fewer nodes. The lower lines correspond to larger amounts of redundant information on more nodes; this provides a wider choice of nodes from which the originator can read the

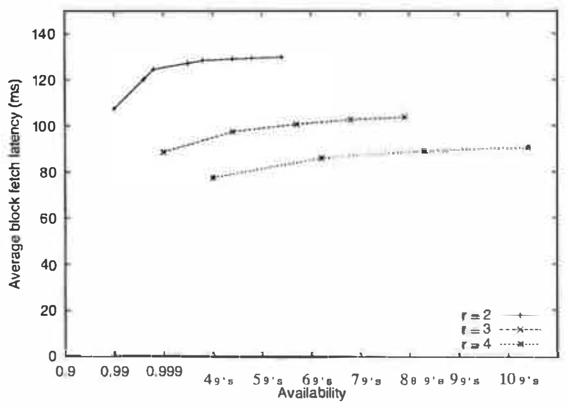


Figure 9: The relationship between read latency and block availability. The different lines correspond to different redundancy factors of $r = 2, 3, 4$. The data points on each line (starting from the left) correspond to reading $m = 1, 2, 3...$ fragments out of $r \times m$ total fragments. The x-axis value is computed from Equation 1 with the per-node availability p_0 set to 0.9, while the y-axis value is the simulated block fetch time (not including lookup time).

```
$a$.lookup($q$, $k$, $d$):
  $overlap = \{n' \mid n' \in \text{succlist}_a \wedge n' > k\}$
  if $|overlap| \geq d$ then
    return $overlap$ to the originator $q$
  else if $overlap \neq \emptyset$ then
    $t$ = {the $s - d$ nodes in succlist$_a$ immediately preceding $k$} $\cup$ $overlap$
    $b = t_i \in t$ s.t. dist$(a, t_i)$ is minimized
    if $b \in overlap$ then
      $t$ = $b$.get_succlist()
      $u$ = merger of $t$ and $overlap$ to produce $k$ first $d$ successors
      return $u$ to the originator $q$
    else
      return $b$.lookup($q$, $k$, $d$)
  else
    $b$ = closestpred(lookupfinger, $k$)
    return $b$.lookup($q$, $k$, $d$)
```

Figure 10: Recursive lookup that returns at least d fragments of key k to sender q. Each node's successor list contains s nodes.

data, which increases the probability that it can read from nearby nodes, and lowers the fetch latency.

The best trade-off between replication and coding is dependent on the workload: a read-intensive workload will experience lower latency with replication, while a write-intensive workload will consume less network bandwidth with coding. DHash++ uses IDA coding with $m = 7$ and $l = 14$. The number seven is selected so that a fragment for an 8 KB block will fit in a single 1500-byte packet, which is important for UDP-based transport. The originator uses Vivaldi (Section 2.3) to predict the latency to the successors.

4.5 Integrating routing and fetching

So far the design of the DHT lookup algorithm and the design of the final data server-selection have been considered separately. One problem with this approach is that obtaining the complete list of a key's s successors requires that the originator contact the key's predecessor, which Section 4.3 observed was expensive because the final lookup steps can take little advantage of proximity routing. However, of the s successors, only the first l immediate successors store the fragments for the key's data block. Furthermore, fragments from any m of these successors are sufficient to reconstruct the block. Each of the $s - m$ predecessor nodes of the key has a successor list that contains m successors. Thus the lookup could stop early at any of those predecessors, avoiding the expensive hop to the predecessor; Pastry/PAST uses a similar technique [38].

However, this design choice decreases the lookup time at the expense of data fetch latency, since it decreases the number of successors (and thus fragments) that the originator can choose from. Once the recursive lookup has reached a node n_1 whose successor list overlaps the key, n_1 is close enough to be the penultimate hop in the routing. By forwarding the query to the closest node n_2 in its successor list that can return enough nodes, n_1 can ensure that the next hop will be the last hop. There are two cases — if n_2 is past the key, then n_1 must directly retrieve n_2's successor list and merge it with its own overlapping nodes to avoid overshooting. Otherwise, n_1 can simply hand-off the query to n_2 who will have enough information to complete the request.

Figure 10 shows the pseudo-code for this final version of the DHash++ lookup algorithm. The d argument indicates how many successors the caller would like. d must be at least as large as m, while setting d to l retrieves the locations of all fragments.

The final latency design decision is the choice of d. A large value forces the lookup to take more hops, but yields more choice for the data fetch and thus lower fetch latency; while a small d lets the lookup finish sooner but yields higher fetch latency. Figure 11 explores this trade-off. It turns out that the cost of a higher d is low, since the lookup algorithm in Figure 10 uses only nearby nodes as the final hops, while the decrease in fetch time by using larger d is relatively large. Thus setting $d = l$ is the best

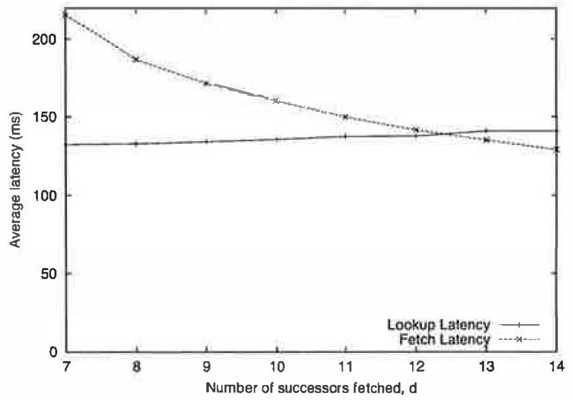


Figure 11: Simulated lookup and fetch time as a function of the d parameter in Figure 10. Larger d causes the lookup to take more hops and gather more successors; the extra successors decrease the fetch latency by providing more choice of nodes to fetch from. For comparison, the average lookup and fetch times that result from always contacting the predecessor are 224 and 129 milliseconds, respectively.

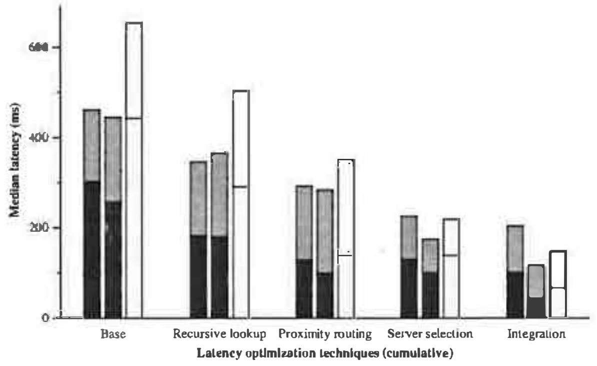


Figure 12: The cumulative effect of successive performance optimizations on the median latency of a DHash++ data fetch. The leftmost bar in each triple shows the time on our PlanetLab implementation (copied from Figure 4). The middle bar was produced by the simulator using a latency matrix measured between PlanetLab hosts. The rightmost bar corresponds to simulations of 2048 nodes using the King latency matrix. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data.

policy.

4.6 Summary

Figure 12 summarizes the cumulative effect of the design decisions explored in this section. The leftmost bar in each triple shows the median time on our PlanetLab implementation (copied from Figure 4). The middle bar was produced by the simulator using a latency matrix measured between PlanetLab hosts. The dark portion of each bar shows the lookup time, and the light portion shows the time taken to fetch the data. Although the simulator results do not match the PlanetLab results exactly, the trends are the same. The results differ because the simulator uses inter-host delays measured between a slightly different set of PlanetLab nodes than were used for the implementation experiments, and at a different time.

The rightmost bar corresponds to simulations of 2048 nodes using the King latency matrix. The absolute numbers are larger than for the PlanetLab results, and perhaps more representative of the Internet as a whole, because the King data set includes a larger and more diverse set of nodes. Again, the overall trends are the same.

5 Achieving high throughput

Some applications, such as Usenet article storage, need to store or retrieve large amounts of data in a DHT. If data movement is to be fast, the DHT must make efficient use of the underlying network resources. The DHT must keep enough data in flight to cover the network's delay-bandwidth product, stripe data over multiple slow access links in parallel, and recover in a timely fashion from packet loss. The DHT must also provide congestion control in order to avoid unnecessary re-transmissions and to avoid overflowing queues and forcing packet loss. These goals are similar to those of traditional unicast transport protocols such as TCP [21], but with the additional requirement that the solution function well when the data is spread over a large set of servers.

This section presents two different designs, then compares their efficiency when implemented in DHash++ on the RON test-bed. We focus here on bulk fetch operations rather than insert operations.

5.1 TCP transport

Perhaps the simplest way for a DHT to manage its consumption of network resources is to use TCP. Because TCP imposes a start-up latency, requires time to acquire good timeout and congestion window size estimates, and consumes host state that limits the number of simultaneous connections, it makes the most sense for a DHT to maintain a relatively small number of long-running TCP connections to its neighbors and to arrange that communication only occur between neighbors in the DHT overlay. This arrangement provides congestion control without burdening the DHT with its implementation. Several systems use this approach (e.g., [34]), some with slight modifications to avoid exhausting the number of file descriptors. For example, Tapestry uses a user-level re-implementation of TCP without in-order delivery [47].

Restricting communication to the overlay links means that all lookups and data movement must be recursive: iterative lookups or direct movement of data would not be able to use the persistent inter-neighbor TCP connections. Section 4.2 showed that recursive lookups work well. However, recursive data movement requires that each block of data be returned through the overlay rather than directly. This recursive return of data causes it to be sent into and out of each hop's Internet access link, potentially increasing latency and decreasing useful throughput. In addition, hiding the congestion control inside TCP limits the options for the design of the DHT's failure recovery algorithms, as well as making it hard for the DHT to control its overall use of network resources. Section 5.3 shows performance results that may help in deciding whether the convenience of delegating congestion control to TCP outweighs the potential problems.

DHash++ allows the option to use TCP as the transport. Each node keeps a TCP connection open to each of its fingers, as well as a connection to each node in its successor list. DHash++ forwards a get request recursively through neighbors' TCP connections until the request reaches a node whose successor list includes a sufficient number of fragments (as in Section 4.5). That node fetches fragments in parallel over the connections to its successors, trying the most proximate successors first. It then re-constructs the block from the fragments and sends the block back through the reverse of the route that the request followed. Pond [34] moves data through the Tapestry overlay in this way.

5.2 STP transport

At the other extreme, a DHT could include its own specialized transport protocol in order to avoid the problems with TCP transport outlined above. This approach allows the DHT more freedom in which nodes it can contact, more control over the total load it places on the network, and better integration between the DHT's failure handling and packet retransmission.

DHash++ allows the option to use a specialized transport called the Striped Transport Protocol (STP). STP allows nodes to put and get data directly to other nodes, rather than routing the data through multiple overlay hops. STP does not maintain any per-destination state; instead, all of its decisions are based on aggregate measurements of recent network behavior, and on Vivaldi latency predictions. STP's core mechanism is a TCP-like congestion window controlling the number of concurrent outstanding RPCs.

While STP borrows many ideas from TCP, DHT data transfers differ in important ways from the unicast transfers that TCP is designed for. Fetching a large quantity of DHT data involves sending lookup and get requests to many different nodes, and receiving data fragments from many nodes. There is no steady "ACK clock" to pace new data, since each RPC has a different destination. The best congestion window size (the number of outstanding RPCs to maintain) is hard to define, because there may be no single delay and thus no single bandwidth-delay product. Quick recovery from lost packets via fast retransmit [41] may not be possible because RPC replies are not likely to arrive in order. Finally, averaging RPC round-trip times to generate time-out intervals may not work well because each RPC has a different destination.

The rest of this section describes the design of STP.

5.2.1 STP window control

Each DHash++ server controls all of its network activity with a single instance of STP. STP maintains a window of outstanding UDP RPCs: it only issues a new RPC when an outstanding RPC has completed. STP counts both DHT lookup and data movement RPCs in the window.

STP maintains a current window size w in a manner similar to that of TCP [21, 9]. When STP receives an RPC reply, it increases w by $1/w$; when an RPC times out, STP halves w.

STP actually keeps $3w$ RPCs in flight, rather than w. Using w would cause STP to transfer data significantly slower than a single TCP connection: lookup RPCs carry less data than a typical TCP packet, STP has nothing comparable to TCP's cumulative acknowledgments to mask lost replies, STP's retransmit timers are more conservative than TCP's, and STP has no mechanism analogous to TCP's fast retransmit. The value 3 was chosen empirically to cause STP's network use to match TCP's.

5.2.2 Retransmit timers

Lost packets have a large negative impact on DHash++ throughput because each block transfer is preceded by a multi-RPC lookup; even a modest packet loss rate may routinely stall the advancement of the window. Ideally STP would choose timeout intervals slightly larger than the true round trip time, in order to to waste the minimum amount of time. This approach would require a good RTT predictor. TCP predicts the RTT using long-term measurements of the average and standard deviation of per-packet RTT [21]. STP, in contrast, cannot count on sending repeated RPCs to the same destination to help it characterize the round-trip time. In order for STP to perform well in a large DHT, it must be able to predict the RTT before it sends even one packet to a given destination.

STP uses Vivaldi latency predictions to help it choose the retransmit time-out interval for each RPC. However, Vivaldi tends to under-predict network delays because it does not immediately account for current network queuing delays or CPU processing time at each end. Since under-predicting the latency of an RPC is costly (a spurious loss detection causes a halving of the current window) STP adjusts the Vivaldi prediction before using it. STP characterizes the errors that Vivaldi makes by keeping a moving average of the difference between each successful RPC's round-trip time and the Vivaldi prediction. STP keeps this average over all RPCs, not per-destination. STP chooses an RPC's retransmission interval in milliseconds as follows:

$$RTO = v + 6 \times \alpha + 15 \qquad (2)$$

where v is the Vivaldi-predicted round trip time to the destination and α is the average error. The weight on the α term was chosen by analyzing the distribution of RPC delays seen by a running node; the chosen timers produce less than 1 percent spurious retransmissions with approximately three times less over-prediction in the case of a loss than a conservative (1 second) timer. This formula assumes that Vivaldi's errors are normally distributed; adding a constant times the error corresponds to sampling a low percentile of the error distribution. The constant α plays a part similar to the measured RTT deviation in the TCP retransmit timer calculation.

The constant term in Equation 2 (15 ms) is necessary to avoid retransmissions to other virtual nodes on the same host; Vivaldi predicts small latencies to the local node, but under high load the observed delay is as much as 15 ms. This term prevents those retransmissions without adding significantly to over-prediction for distant nodes.

5.2.3 Retransmit policy

When an STP retransmit timer expires, STP notifies the application (DHash++) rather than re-sending the RPC. This gives DHash++ a chance to re-send the RPC to a different destination. DHash++ re-sends a lookup RPC to the finger that is next-closest in ID space, and re-sends a fragment fetch RPC to the successor that is next-closest in predicted latency. This policy helps to avoid wasting time sending RPCs to nodes that have crashed or have overloaded access links.

DHash++ uses a separate background stabilization process to decide whether nodes in the finger table or successor list have crashed; it sends periodic probe RPCs and decides a node is down only when it fails to respond to many probes in a row.

5.3 Performance comparison

This section presents measurements comparing the latency and throughput of the TCP transport implementation to the STP implementation when run on the RON test-bed. We used 26 RON nodes, located in the United States and Europe. Each physical RON node is located in a different machine room and ran 4 copies of DHash++.

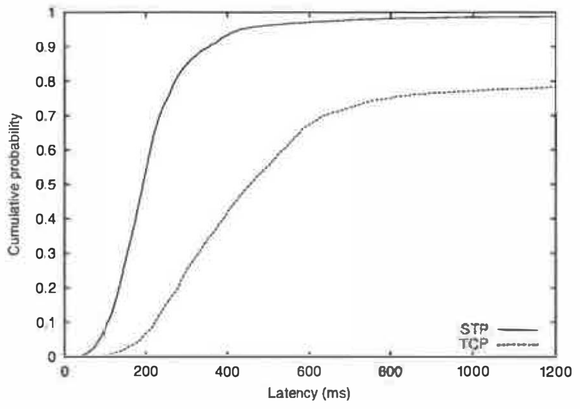


Figure 13: Distribution of individual 8192-byte fetch latencies on RON.

The average inter-node round-trip time is 75 ms, and the median is 72 ms (these reflect the multiple copies of DHash++ per host).

5.3.1 Fetch latency

Figure 13 shows the distribution of individual block fetch latencies on RON. The numbers are derived from an experiment in which each node in turn fetched a sequence of randomly chosen blocks; at any given time only one fetch was active in the DHT. The median fetch time was 192 ms with STP and 447 ms with TCP. The average number of hops required to complete a lookup was 3.

The STP latency consists of approximately 3 one-way latencies to take the lookup to the predecessor, plus one one-way latency to return the lookup reply to the originator. The parallel fetch of the closest seven fragments is limited by the latency to the farthest fragment, which has median latency (see Section 4.4). Thus the total expected time is roughly $4 \times 37.5 + 72 = 222$; the actual median latency of 192 ms is probably less due to proximity routing of the lookup.

The TCP latency consists of the same three one-way latencies to reach the predecessor, then a median round-trip-time for the predecessor to fetch the closest seven fragments, then the time required to send the 8 KB block over three TCP connections in turn. If the connection uses slow-start, the transfer takes 2.5 round trip times (there's no need to wait for the last ACK); if not, just half a round-trip time. A connection only uses slow-start if it has been idle for a second or more. The connection from the first hop back to the originator is typically not idle, because it has usually been used by a recent fetch in the experiment; the other connections are much more likely to use slow start. Thus the latency should range from 340 ms if there was no slow-start, to 600 ms if two of the hops used slow-start. The measured time of 447 ms falls in this range. This analysis neglects the transmission time of an 8 KB block

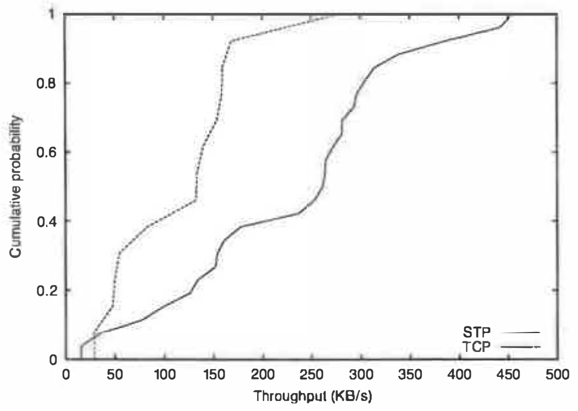


Figure 14: Distribution of average throughput obtained by different RON nodes during 4 megabyte transfers.

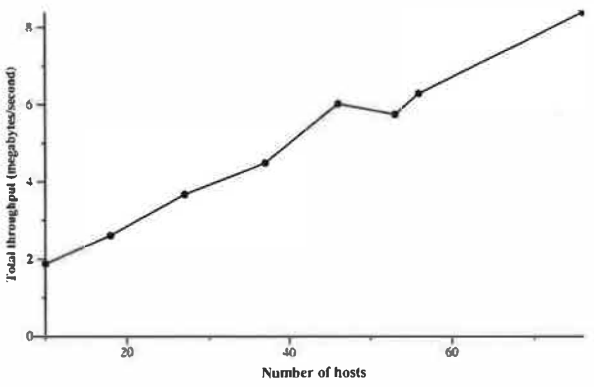


Figure 15: The effect of system size on total throughput obtainable. Each point represents an experiment with DHash++ running at x sites on the RON and PlanetLab test-beds. Each site reads 1000 8 KB blocks; the aggregate throughput of the system in steady state is reported. This throughput increases as additional capacity (in the form of additional sites) is added to the system.

(about 131 ms at 1 Mb/s).

5.3.2 Single-client fetch throughput

Figure 14 shows the distribution of fetch throughput achieved by different RON nodes when each fetches a long sequence of blocks from DHash++. The application maintains 64 one-block requests outstanding to its local DHash++ server, enough to avoid limiting the size of STP's congestion window.

Using TCP transport, the median node achieved a throughput of 133 KB/s. The minimum and maximum throughputs were 29 and 277 KB/s. Both the median throughput and the range of individual node throughputs are higher when using STP: the median was 261 KB/s, and throughputs ranged from 15 to 455 KB/s. The TCP transport has lower throughput because it sends each block back through each node on the recursive route, and thus is more likely than STP to send a block through a slow access link. About half of the three-hop routes pass through one of the RON sites with sub-one-megabit access links. STP sends coded fragments directly to the node originating the request, and thus each fragment encounters fewer slow links.

To characterize the effectiveness of STP in utilizing available resources we consider the expected throughput of a DHash++ system. Assuming an STP window large enough to keep all links busy, a node can fetch data at a rate equal to the slowest access link times the number of nodes, since the blocks are spread evenly over the nodes.

The slowest site access link in RON has a capacity of about 0.4 Mb/s. With 26 nodes one would expect $0.4 \times 26 = 10.4$ Mb/s or 1.3 MB/s total throughput for a fetching site not limited by its own access link. STP achieves less than half of this throughput at the fastest site. The reason appears to be that STP has difficulty maintaining a large window in the face of packet loss, which averages about 2 percent in these tests.

5.3.3 Scale

This section evaluates the ability of STP and DHash++ to take advantage of additional resources. As the number of nodes grows, more network capacity (in the form of additional access links) is added to the system. Figure 15 shows the total throughput for an N-node DHT when all N nodes simultaneously read a large number of blocks, as a function of N. The experiments were run on the combined PlanetLab and RON test-beds. The slowest access link was that of a node in Taiwan, which was able to send at 200 KB/s to sites in the US. The observed throughput corresponds to our throughput prediction: the total throughputs scales with the number of sites. The first data point consists of ten sites experiencing an aggregate of ten times the bandwidth available at the slowest site.

A similar experiment run using 150 machines but at 70 unique sites (many PlanetLab sites are home to more than one node) produces a peak throughput of 12.8 MB/s. As more machines and DHash++ nodes are added to each site, that site gains a proportionally greater share of that site's link bandwidth and the system's aggregate bandwidth increases.

6 Related work

The primary contribution of this paper is exploring a large set of design decisions for DHTs in the context of a single, operational system. This exploration of design decisions emerged from an effort to understand a number of recent DHT-like systems with different designs, including OceanStore/Pond [22, 34, 43], CFS [13], Overnet [28, 32], PAST [6, 37, 38] FarSite [1], and Pastiche [11].

Rhea *et al.* did a black-box comparison of the implementations of several structured lookup systems (Chord, Pastry, Tapestry) and found that the use of proximity information reduced lookup latency, especially for lookups destined to nearby hosts [36].

Gummadi *et al.* studied the impact of routing geometry on resilience and proximity [16]. They found that flexibility in the routing geometry in general improved the ability of the system to find good neighbors. We extend their findings with additional analysis in simulation and with actual measurements to better understand the performance gains seen when using proximity.

A number of recent papers discuss design ideas related to networks with churn [5, 23, 35], some of which are used by DHash++. These ideas include integrated transport systems that quickly detect node failure and use alternate routes after timeouts.

7 Conclusions and future work

This paper has presented a series of design decisions faced by DHTs that store data, discussed the design options and how they interact, and compared a number of variant designs using simulations and measurements of an implementation running on PlanetLab and RON. The paper proposed techniques that taken together together reduce fetch latency by a factor of two and allow efficient bulk throughput.

The list of design decisions is not exhaustive, and future work will analyze a wider range of DHT designs and behavior such as the relationship between lookup robustness and performance, the latency and throughput of DHT writes, the handling of data movement required when nodes join and leave, data layout policies, the effects of block size, and the tradeoffs involved in use of constant-hop-count lookup protocols.

The simulator and DHash++ are publically available from `http://project-iris.net`.

Acknowledgements

We would like to thank Chuck Blake for his help with data analysis, Thomer Gil, Jeremy Stribling and Russ Cox for their help writing the simulator, Peter Druschel and the anonymous reviewers for their helpful comments, and David Andersen and the PlanetLab project for the RON and PlanetLab test-beds.

This paper is dedicated to the memory of Josh Cates.

References

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 5th OSDI* (Dec. 2002).

[2] Andersen, D., Balakrishnan, H., Kaashoek, M. F., and Morris, R. Resilient overlay networks. In *Proc. of the 18th ACM SOSP* (Chateau Lake Louise, Banff, Canada, October 2001).

[3] Anderson, R. J. The eternity service. In *Proc. of the 1996 Pragocrypt* (1996).

[4] Blake, C., and Rodrigues, R. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of the 9th Workshop on Hot Topics in Operating Systems* (May 2003).

[5] Castro, M., Costa, M., and Rowstron, A. Performance and dependability of structured peer-to-peer overlays. Tech. Rep. MSR-TR-2003-94, Microsoft Research, December 2003.

[6] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in peer-to-peer overlay networks. Tech. Rep. MSR-TR-2002-82, Microsoft Research, June 2002.

[7] Cates, J. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[8] Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., and Yianilos, P. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital libraries* (Berkeley, CA, Aug. 1999), pp. 28–37.

[9] Chiu, D.-M., and Jain, R. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems 17* (1989), 1–14.

[10] Costa, M., Castro, M., Rowstron, A., and Key, P. PIC: Practical Internet coordinates for distance estimation. In *24th International Conference on Distributed Computing Systems* (Tokyo, Japan, March 2004).

[11] Cox, L. P., and Noble, B. D. Pastiche: making backup cheap and easy. In *Proc. of the 5th OSDI* (Dec. 2002).

[12] Cox, R., Dabek, F., Kaashoek, F., Li, J., and Morris, R. Practical, distributed network coordinates. In *Proc. of the Second workshop on Hot Topics in Networks (HotNets-II)* (Nov. 2003).

[13] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM SOSP* (Oct. 2001).

[14] Ganesh, A., Kermarrec, A.-M., and Massoulie, L. HiSCAMP: self-organising hierarchical membership protocol. In *Proc. of the 10th European ACM SIGOPS workshop* (Sept. 2002).

[15] Gummadi, K., Saroiu, S., and Gribble, S. D. King: Estimating latency between arbitrary Internet end hosts. In *Proc. of the 2002 SIGCOMM Internet Measurement Workshop* (Marseille, France, Nov. 2002).

[16] Gummadi, K. P., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., and Stoica, I. The impact of DHT routing geometry on resilience and proximity. In *Proc. of the 2003 ACM SIGCOMM* (Karlsruhe, Germany, Aug. 2003).

[17] Gupta, A., Liskov, B., and Rodrigues, R. One hop lookups for peer-to-peer overlays. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems* (May 2003).

[18] Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. of the 2nd IPTPS* (Feb. 2003).

[19] Hand, S., and Roscoe, T. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of the 1st IPTPS* (Mar. 2001).

[20] IYER, S., ROWSTRON, A., AND DRUSCHEL, P. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*. (July 2002).

[21] JACOBSON, V. Congestion avoidance and control. In *Proc. of the ACM SIGCOMM* (Aug. 1988).

[22] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)* (Boston, MA, Nov. 2000), pp. 190–201.

[23] LI, J., STRIBLING, J., MORRIS, R., KAASHOEK, M. F., AND GIL, T. DHT routing tradeoffs in networks with churn. In *Proc. of the 3rd IPTPS* (Feb. 2004).

[24] LIM, H., HOU, J., AND CHOI, C.-H. Constructing an Internet coordinate system based on delay measurement. In *Proc. of the 2003 SIGCOMM Internet Measurement Conference* (Oct. 2003).

[25] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of the 1st IPTPS* (Mar. 2002).

[26] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *Proc. of the 5th OSDI* (Dec. 2002).

[27] NG, T. S. E., AND ZHANG, H. Predicting Internet network distance with coordinates-based approaches. In *Proc. of the 2002 IEEE Infocom* (June 2002).

[28] Overnet. `http://www.overnet.com/`.

[29] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A blueprint for introducing disruptive technology into the Internet. In *Proc. of HotNets-1* (October 2002). `http://www.planet-lab.org`.

[30] PIAS, M., CROWCROFT, J., WILBUR, S., HARRIS, T., AND BHATTI, S. Lighthouses for scalable distributed location. In *Proc. of the 2nd IPTPS* (Feb. 2003).

[31] RABIN, M. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (Apr. 1989), 335–348.

[32] RANJITA BHAGWAN, S. S., AND VOELKER, G. Understanding availability. In *Proc. of the 2nd IPTPS* (Feb. 2003).

[33] RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. Topologically-aware overlay construction and server selection. In *Proceedings of Infocom 2002* (2002).

[34] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (Apr. 2003).

[35] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. Tech. Rep. UCB/CSD-3-1299, UC Berkeley, Computer Science Division, Dec. 2003.

[36] RHEA, S., ROSCOE, T., AND KUBIATOWICZ, J. Structured peer-to-peer overlays need application-driven benchmarks. In *Proc. of the 2nd IPTPS* (Feb. 2003).

[37] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distr ibuted Systems Platforms (Middleware 2001)* (Nov. 2001).

[38] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM SOSP* (Oct. 2001).

[39] SHAVITT, Y., AND TANKEL, T. Big-bang simulation for embedding network distances in Euclidean space. In *Proc. of IEEE Infocom* (April 2003).

[40] SIT, E., DABEK, F., AND ROBERTSON, J. UsenetDHT: A low overhead usenet server. In *Proc. of the 3rd IPTPS* (Feb. 2004).

[41] STEVENS, W. R. RFC2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Tech. rep., Internet Assigned Numbers Authority, 1997.

[42] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM* (San Diego, Aug. 2001). An extended version appears in ACM/IEEE Trans. on Networking.

[43] STRIBLING, J. Optimizations for locality-aware structured peer-to-peer overlays. In *Proc. of the 1st IRIS Student Workshop* (Cambridge, MA, Aug. 2003).

[44] VOULGARIS, S., AND VAN STEEN., M. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *Proc. of the 14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003)* (Oct. 2003).

[45] WALFISH, M., BALAKRISHNAN, H., AND SHENKER, S. Untangling the web from DNS. In *Proc. of the 1st NSDI* (Mar. 2004).

[46] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the 1st IPTPS* (Mar. 2002).

[47] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004).

Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays

Venugopalan Ramasubramanian and Emin Gün Sirer
Dept. of Computer Science,
Cornell University,
Ithaca NY 14853
{ramasv, egs}@cs.cornell.edu

Abstract

Structured peer-to-peer hash tables provide decentralization, self-organization, failure-resilience, and good worst-case lookup performance for applications, but suffer from high latencies (O($logN$)) in the average case. Such high latencies prohibit them from being used in many relevant, demanding applications such as DNS. In this paper, we present a proactive replication framework that can provide constant lookup performance for common Zipf-like query distributions. This framework is based around a closed-form optimal solution that achieves O(1) lookup performance with low storage requirements, bandwidth overhead and network load. Simulations show that this replication framework can realistically achieve good latencies, outperform passive caching, and adapt efficiently to sudden changes in object popularity, also known as flash crowds. This framework provides a feasible substrate for high-performance, low-latency applications, such as peer-to-peer domain name service.

1 Introduction

Peer-to-peer distributed hash tables (DHTs) have recently emerged as a building-block for distributed applications. *Unstructured* DHTs, such as Freenet and the Gnutella network [1, 5], offer decentralization and simplicity of system construction, but may take up to O(N) hops to perform lookups in networks of N nodes. *Structured* DHTs, such as Chord, Pastry, Tapestry and others [14, 17, 18, 21, 22, 24, 28], are particularly well-suited for large scale distributed applications because they are self-organizing, resilient against denial-of-service attacks, and can provide O(log N) lookup performance. However, for large-scale, high-performance, latency-sensitive applications, such as the domain name service (DNS) and the world wide web, this logarithmic performance bound translates into high latencies. Previous work on serving DNS using a peer-to-peer lookup service concluded that, despite their desirable properties, structured DHTs are unsuitable for latency-sensitive applications due to their high lookup costs [8].

In this paper, we describe how proactive replication can be used to achieve constant lookup performance efficiently on top of a standard O(log N) peer-to-peer distributed hash table for certain commonly-encountered query distributions. It is well-known that the query distributions of several popular applications, including DNS and the web, follow a power law distribution [2, 15]. Such a well-characterized query distribution presents an opportunity to optimize the system according to the expected query stream. The critical insight in this paper is that, for query distributions based on a power law, *proactive* (model-driven) replication can enable a DHT system to achieve a small constant lookup latency on average. In contrast, we show that common techniques for *passive* (demand-driven) replication, such as caching objects along a lookup path, fail to make a significant impact on the average-case behavior of the system.

We outline the design of a replication framework, called Beehive, with the following goals:

- **High Performance**: Enable O(1) average-case lookup performance, effectively decoupling the performance of peer-to-peer DHT systems from the size of the network. Provide O(log N) worst-case lookup performance.
- **High Scalability**: Minimize the background traffic in the network to reduce aggregate network load and per-node bandwidth consumption. Keep the memory and disk space requirements at each peer to a minimum.
- **High Adaptivity**: Adjust the performance of the system to the popularity distribution of objects. Respond quickly when object popularities change, as with flash crowds and the "slashdot effect."

Beehive achieves these goals through efficient proactive replication. By proactive replication, we mean actively propagating copies of objects among the nodes participating in the network. There is a fundamental tradeoff between replication and resource consumption: more copies of an object will generally improve lookup performance at the cost of space, bandwidth and aggregate network load.

Beehive performs this tradeoff through an informed analytical model. This model provides a closed-form, optimal solution that guarantees O(1), constant-time lookup performance with the minimum number of object replicas. The particular constant C targeted by the system is tunable. Beehive enables the system designer to specify a fractional value. Setting C to a fractional value, such as 0.5, ensures that 50% of queries will be satisfied at the source, without any additional network hops. Consequently, Beehive implements a sub-one hop hash-table. The value of C can be adjusted dynamically to meet real-time performance goals.

Beehive uses the minimal number of replicas required to achieve a targeted performance level. Minimizing replicas reduces storage requirements at the peers, lowers bandwidth consumption and load in the network, and enables cache-coherent updates to be performed efficiently. Beehive uses low-overhead protocols for tracking, propagating and updating replicas. Finally, Beehive leverages the structure of the underlying DHT to update objects efficiently at runtime, and guarantees that subsequent lookups will return the latest copy of the object.

While this paper describes the Beehive proactive replication framework in its general form, we use the domain name system as a driving application. Several shortcomings of the current, hierarchical structure of DNS makes it an ideal application candidate for Beehive. First, DNS is highly latency-sensitive, and poses a significant challenge to serve efficiently. Second, the hierarchical organization of DNS leads to a disproportionate amount of load being placed at the higher levels of the hierarchy. Third, the higher nodes in the DNS hierarchy serve as easy targets for distributed denial-of-service attacks and form a security vulnerability for the entire system. Finally, nameservers required for the internal leaves of the DNS hierarchy incur expensive administrative costs, as they need to be manually administered and secured. Peer-to-peer DHTs address all but the first critical problem; we show in this paper that Beehive's replication strategy can address the first.

We have implemented a prototype Beehive-based DNS server on Pastry [22]. We envision that the DNS nameservers that are currently used to serve small, dedicated portions of the naming hierarchy would form a Beehive network and collectively serve the namespace. While we use DNS as a guiding application and demonstrate that serving DNS with DHT is feasible, we note that a full treatment of the implementation of an alternative peer-to-peer DNS system is beyond the scope of this paper, and focus instead on the general-purpose Beehive framework for proactive replication. The framework is sufficiently general to achieve O(1) lookup performance in other settings, including web caching, where the query distribution follows a power law.

Overall, this paper describes the design of a replication framework that enables constant lookup performance in structured DHTs for common query distributions, applies it to a P2P DNS implementation, and makes the following contributions. First, it proposes proactive replication of objects and provides a closed-form analytical solution for the optimal number of replicas needed to achieve O(1) lookup performance. The storage, bandwidth and load placed on the network by this scheme are modest. In contrast, we show that simple caching strategies based on passive replication incur large ongoing costs. Second, it outlines the design of a complete system based around this analytical model. This system is layered on top of Pastry, an existing peer-to-peer substrate. It includes techniques for estimating the requisite inputs for the analytical model, mechanisms for replica distribution and deletion, and fast update propagation. Finally, it presents results from a prototype implementation of a peer-to-peer DNS service to show that the system achieves good performance, has low overhead, and can adapt quickly to flash crowds. In turn, these approaches enable the benefits of P2P systems, such as self-organization and resilience against denial of service attacks, to be applied to latency-sensitive applications.

The rest of this paper is organized as follows. Section 2 provides a broad overview of our approach and describes the storage and bandwidth-efficient replication components of Beehive in detail. Section 3 describes our implementation of Beehive over Pastry. Section 4 presents the results and expected benefits of using Beehive to serve DNS queries. Section 5 surveys different DHT systems and summarizes other approaches to caching and replication in peer-to-peer systems Section 6 describes future work and Section 7 summarizes our contributions.

2 The Beehive System

Beehive is a general replication framework that operates on top of any DHT that uses prefix-routing [19], such as Chord [24], Pastry [22], Tapestry [28], and Kademlia [18]. Such DHTs operate in the following manner. Each node has a unique randomly assigned identifier in a circular identifier space. Each object also has a

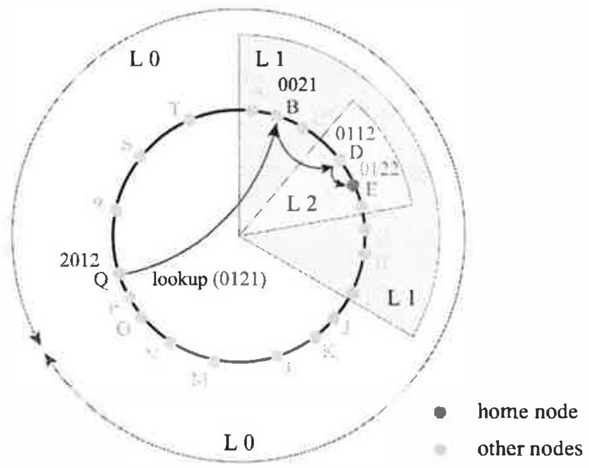


Figure 1: **This figure illustrates the levels of replication in Beehive. A query for object 0121 takes three hops from node Q to node E, the home node of the object. By replicating the object at level 2, that is at D and F, the query latency can be reduced to two hops. In general, an object replicated at level i incurs at most i hops for a lookup.**

unique randomly selected identifier, and is stored at the node whose identifier is closest to its own, called the *home node*. Routing is performed by successively matching a prefix of the object identifier against node identifiers. Generally, each step in routing takes the query to a node that has one more matching prefix than the previous node. A query traveling k hops reaches a node that has k matching prefixes[1]. Since the search space is reduced exponentially, this query routing approach provides O($log_b N$) lookup performance on average, where N is the number of nodes in the DHT and b is the base, or fanout, used in the system.

The central observation behind Beehive is that the length of the average query path will be reduced by one hop when an object is proactively replicated at all nodes logically preceding that node on all query paths. For example, replicating the object at all nodes one hop prior to the home-node decreases the lookup latency by one hop. We can apply this iteratively to disseminate objects widely throughout the system. Replicating an object at all nodes k hops or lesser from the home node will reduce the lookup latency by k hops. The Beehive replication mechanism is a general extension of this observation to find the appropriate amount of replication for each object based on its popularity.

Beehive controls the extent of replication in the system by assigning a *replication level* to each object. An object at level i is replicated on all nodes that have at least i matching prefixes with the object. Queries to objects replicated at level i incur a lookup latency of at most i hops. Objects stored only at their home nodes are at level $log_b N$, while objects replicated at level 0 are cached at all the nodes in the system. Figure 1 illustrates the concept of replication levels.

The goal of Beehive's replication strategy is to find the minimal replication level for each object such that the average lookup performance for the system is a constant C number of hops. Naturally, the optimal strategy involves replicating more popular objects at lower levels (on more nodes) and less popular objects at higher levels. By judiciously choosing the replication level for each object, we can achieve constant lookup time with minimal storage and bandwidth overhead.

Beehive employs several mechanisms and protocols to find and maintain appropriate levels of replication for its objects. First, an analytical model provides Beehive with closed-form optimal solutions indicating the appropriate levels of replication for each object. Second, a monitoring protocol based on local measurements and limited aggregation estimates relative object popularity, and the global properties of the query distribution. These estimates are used, independently and in a distributed fashion, as inputs to the analytical model which yields the locally desired level of replication for each object. Finally, a replication protocol proactively makes copies of the desired objects around the network. The rest of this section describes each of these components in detail.

2.1 Analytical Model

In this section, we provide a model that analyzes Zipf-like query distributions and provides closed-form optimal replication levels for the objects in order to achieve constant average lookup performance with low storage and bandwidth overhead.

In Zipf-like, or power law, query distributions, the number of queries to the i^{th} most popular object is proportional to $i^{-\alpha}$, where α is the parameter of the distribution. The query distribution has a heavier tail for smaller values of the parameter α. A Zipf distribution with parameter 0 corresponds to a uniform distribution. The total number of queries to the most popular m objects, $Q(m)$, is approximately $\frac{m^{1-\alpha}-1}{1-\alpha}$ for $\alpha \neq 1$, and $Q(m) \simeq ln(m)$ for $\alpha = 1$.

Using the above estimate for the number of queries received by objects, we pose an optimization problem to minimize the total number of replicas with the constraint that the average lookup latency is a constant C.

Let b be the base of the underlying DHT system, M the number of objects, and N the number of nodes in the system. Initially, all M objects in the system are stored

[1]Strictly speaking, the nodes encountered towards the end of the query routing process may not share progressively more prefixes with the object, but remain numerically close. This detail does not significantly impact either the time complexity of standard DHTs or our replication algorithm. Section 3 discusses the issue in more detail.

only at their home nodes, that is, they are replicated at level $k = log_b N$. Let x_i denote the fraction of objects replicated at level i or lower. From this definition, x_k is 1, since all objects are replicated at level k. Mx_0 most popular objects are replicated at all the nodes in the system.

Each object replicated at level i is cached in N/b^i nodes. $Mx_i - Mx_{i-1}$ objects are replicated on nodes that have exactly i matching prefixes. Therefore, the average number of objects replicated at each node is given by $Mx_0 + \frac{M(x_1 - x_0)}{b} + \cdots + \frac{M(x_k - x_{k-1})}{b^k}$. Consequently, the average per node storage requirement for replication is:

$$M[(1 - \frac{1}{b})(x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k-1}}{b^{k-1}}) + \frac{1}{b^k}] \quad (1)$$

The fraction of queries, $Q(Mx_i)$, that arrive for the most popular Mx_i objects is approximately $\frac{(Mx_i)^{1-\alpha} - 1}{M^{1-\alpha} - 1}$. The number of objects that are replicated at level i is $Mx_i - Mx_{i-1}, 0 < i \leq k$. Therefore, the number of queries that travel i hops is $Q(Mx_i) - Q(Mx_{i-1}), 0 < i \leq k$. The average lookup latency of the entire system can be given by $\sum_{i=1}^{k} i(Q(Mx_i) - Q(Mx_{i-1}))$. The constraint on the average latency is $\sum_{i=1}^{k} i(Q(Mx_i) - Q(Mx_{i-1})) \leq C$, where C is the required constant lookup performance. After substituting the approximation for $Q(m)$, we arrive at the following optimization problem.

$$\text{Minimize } x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k-1}}{b^{k-1}}, \text{ such that} \quad (2)$$

$$x_0^{1-\alpha} + x_1^{1-\alpha} + \cdots + x_{k-1}^{1-\alpha} \geq k - C(1 - \frac{1}{M^{1-\alpha}}) \quad (3)$$

$$\text{and } x_0 \leq x_1 \leq \cdots \leq x_{k-1} \leq 1 \quad (4)$$

Note that constraint 4 effectively reduces to $x_{k-1} \leq 1$, since any optimal solution to the problem with just constraint 3 would satisfy $x_0 \leq x_1 \leq \cdots \leq x_{k-1}$.

We can use the Lagrange multiplier technique to find an analytical closed-form optimal solution to the above problem with just constraint 3, since it defines a convex feasible space. However, the resulting solution may not guarantee the second constraint, $x_{k-1} \leq 1$. If the obtained solution violates the second constraint, we can force x_{k-1} to 1 and apply the Lagrange multiplier technique to the modified problem. We can obtain the optimal solution by repeating this process iteratively until the second constraint is satisfied. However, the symmetric property of the first constraint facilitates an easier analytical approach to solve the optimization problem without iterations.

Assume that in the optimal solution to the problem, $x_0 \leq x_1 \leq \cdots \leq x_{k'-1} < 1$, for some $k' \leq k$, and $x_{k'} = x_{k'+1} = \cdots = x_k = 1$. Then we can restate the optimization problem as follows:

$$\text{Minimize } x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k'-1}}{b^{k'-1}}, \text{ such that} \quad (5)$$

$$x_0^{1-\alpha} + x_1^{1-\alpha} + \cdots + x_{k'-1}^{1-\alpha} \geq k' - C', \quad (6)$$

$$\text{where } C' = C(1 - \frac{1}{M^{1-\alpha}})$$

This leads to the following closed-form solution:

$$x_i^* = [\frac{d^i(k' - C')}{1 + d + \cdots + d^{k'-1}}]^{\frac{1}{1-\alpha}}, \forall 0 \leq i < k' \quad (7)$$

$$x_i^* = 1, \forall k' \leq i \leq k \quad (8)$$

$$\text{where } d = b^{\frac{1-\alpha}{\alpha}}$$

We can derive the value of k' by satisfying the condition that $x_{k'-1} < 1$, that is, $\frac{d^{k'-1}(k' - C')}{1 + d + \cdots + d^{k'-1}} < 1$.

As an example, consider a DHT with base 32, $\alpha = 0.9$, $10,000$ nodes, and $1,000,000$ objects. Applying this analytical method to achieve an average lookup time, C, of one hop yields $k' = 2$, $x_0 = 0.001102$, $x_1 = 0.0519$, and $x_2 = 1$. Thus, the most popular 1102 objects would be replicated at level 0, the next most popular 50814 objects would be replicated at level 1, and all the remaining objects at level 2. The average per node storage requirement of this system would be 3700 objects.

The optimal solution obtained by this model applies only to the case $\alpha < 1$. For $\alpha > 1$, the closed-form solution will yield a level of replication that will achieve the target lookup performance, but the amount of replication may not be optimal because the feasible space is no longer convex. For $\alpha = 1$, we can obtain the optimal solution by using the approximation $Q(m) = \ln m$ and applying the same technique. The optimal solution for this case is as follows:

$$x_i^* = \frac{M^{\frac{-C}{k'}} b^i}{b^{\frac{k'-1}{2}}}, \forall 0 \leq i < k' \quad (9)$$

$$x_i^* = 1, \forall k' \leq i \leq k \quad (10)$$

This analytical solution has three properties that are useful for guiding the extent of proactive replication. First, the analytical model provides a solution to achieve any desired constant lookup performance. The system can be tailored, and the amount of overall replication controlled, for any level of performance by adjusting C over a continuous range. Since structured DHTs preferentially keep physically nearby hosts in their top-level routing tables, even a large value for C can dramatically speed up end-to-end query latencies [4]. Second, for a large class of query distributions ($\alpha \leq 1$), the solution provided by this model achieves the optimal number of object replicas required to provide the desired performance. Minimizing the number of replicas reduces per-node storage requirements, bandwidth consumption and aggregate load on the network. Finally, k' serves as an upper bound for the worst case lookup time for any successful query, since all objects are replicated at least in level k'.

We make two assumptions in the analytical model: all objects incur similar costs for replication, and objects do not change very frequently. For applications such as DNS, which have essentially homogeneous object sizes and whose update-driven traffic is a very small fraction of the replication overhead, the analytical model provides an efficient solution. Applying the Beehive approach to applications such as the web, which has a wide range of object sizes and frequent object updates, may require an extension of the model to incorporate size and update frequency information for each object. A simple solution to standardize object sizes is to replicate object pointers instead of the objects themselves. While effective and optimal, this approach adds an extra hop to C and violates sub-one hop routing.

2.2 Popularity and Zipf-Parameter Estimation

The analytical model described in the previous section requires estimates of the α parameter of the query distribution and the relative popularities of the objects. Beehive employs a combination of local measurement and limited aggregation to keep track of these parameters and adapt the replication appropriately.

Beehive nodes locally measure the number of queries received for each object on that node. For non-replicated objects, the count at the home node reflects the popularity of that object. However, queries for an object replicated at level i are evenly distributed across approximately N/b^i nodes. In order to estimate popularity of such an object as accurately as an unreplicated object, one would need an N/b^i-fold increase in the measurement interval. Since dilating the sampling interval would prevent the system from reacting quickly to changes in object popularity, Beehive aggregates popularity data from multiple nodes to arrive at accurate estimates of object popularity within a relatively short sampling interval.

Aggregation in Beehive takes place periodically, once every *aggregation interval*. Each node A sends to node B in the i^{th} level of its routing table an *aggregation message* containing the number of accesses of each object replicated at level i or lower and having $i+1$ matching prefixes with B. When node B receives these messages from all nodes at level i, it aggregates the access counts and sends the result to all nodes in the $(i+1)^{th}$ level of its routing table. This process allows popularity data to flow towards the home node.

A similar process operates in reverse to disseminate the aggregated totals from the higher levels towards the nodes at the leaves. A node at level $i+1$ sends the latest aggregated estimate of access counts to nodes at level i for that object. This exchange occurs when the higher level node is contacted by lower level node for aggregation. For an object replicated at level i, it takes $2(\log N - i)$ rounds of aggregation to complete the information flow from the leaves up to the home node and back.

The overall Zipf-parameter, α, is also estimated in a similar manner. Each node locally estimates α using linear regression to compute the slope of the best fit line, since a Zipf-like popularity distribution is a straight line in log-scale. Beehive nodes then refine this estimate by averaging it with the local estimates of other nodes they communicate with during aggregation.

There will be fluctuations in the estimation of access frequency and the Zipf parameter due to temporal variations in the query distribution. In order to avoid large discontinuous changes to an estimate, we age the estimate using exponential decay.

2.3 Replication Protocol

Beehive requires a protocol to replicate objects at the replication levels determined by the analytical model. In order to be deployable in wide area networks, the replication protocol should be asynchronous and not require expensive mechanisms such as distributed consensus or agreement. In this section, we describe an efficient protocol that enables Beehive to replicate objects across a DHT.

Beehive's replication protocol uses an asynchronous and distributed algorithm to implement the optimal solution provided by the analytical model. Each node is responsible for replicating an object on other nodes at most one hop away from itself; that is, at nodes that share one less prefix than the current node. Initially, each object is replicated only at the home node at a level $k = log_b N$, where N is the number of nodes in the system and b is the base of the DHT, and shares k prefixes with the object. If an object needs to be replicated at the next level $k-1$, the home node pushes the object to all nodes that share one less prefix with the home node. Each of the level $k-1$ nodes at which the object is currently replicated may independently decide to replicate the object further, and push the object to other nodes that share one less prefix with it. Nodes continue the process of independent and distributed replication until all the objects are replicated at appropriate levels. In this algorithm, nodes

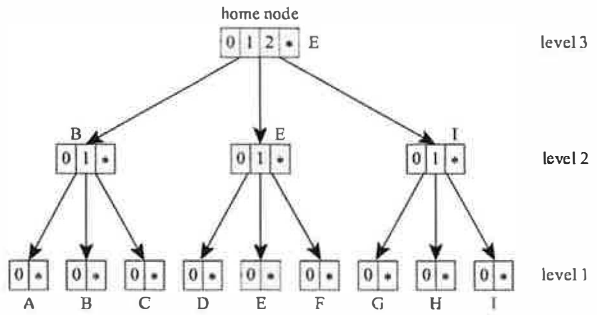


Figure 2: **This figure illustrates how the object 0121 at its home node E is replicated to level 1. For nodes A through I, the numbers indicate the prefixes that match the object identifier at different levels. Each node pushes the object independently to nodes with one less matching digit.**

that share $i+1$ prefixes with an object are responsible for replicating that object at level i, and are called *i level deciding nodes* for that object. For each object replicated at level i at some node A, the i level deciding node is that node in its routing table at level i that has matching $i+1$ prefixes with the object. For some objects, the deciding node may be the node A itself.

This distributed replication algorithm is illustrated in Figure 2. Initially, an object with identifier 0121 is replicated at its home node E at level 3 and shares 3 prefixes with it. If the analytical model indicates that this object should be replicated at level 2, node E pushes the objects to nodes B and I with which it shares 2 prefixes. Node E is the level 2 deciding node for the object at nodes B, E, and I. Based on the popularity of the object, the level 2 nodes B, E, and I may independently decide to replicate the object at level 1. If node B decides to do so, it pushes a copy of the object to nodes A and C with which it shares 1 prefix and becomes the level 1 deciding node for the object at nodes A, B, and C. Similarly, node E may replicate the object at level 1 by pushing a copy to nodes D and F, and node I to G and F.

Our replication algorithm does not require any agreement in the estimation of relative popularity among the nodes. Consequently, some objects may be replicated partially due to small variations in the estimate of the relative popularity. For example in Figure 2, node E might decide not to push object 0121 to level 1. We tolerate this inaccuracy to keep the replication protocol efficient and practical. In the evaluation section, we show that this inaccuracy in the replication protocol does not produce any noticeable difference in performance.

Beehive implements this distributed replication algorithm in two phases, an *analysis phase* and a *replicate phase*, that follow the aggregation phase. During the analysis phase, each node uses the analytical model and the latest known estimate of the Zipf-parameter α to obtain a new solution. Each node then locally changes the replication levels of the objects according to the solution. The solution specifies for each level i, the fraction of objects, x_i that need to be replicated at level i or lower. Hence, $\frac{x_i}{x_{i+1}}$ fraction of objects replicated at level $i+1$ or lower should be replicated at level i or lower. Based on the current popularity, each node sorts all the objects at level $i+1$ or lower for which it is the i level deciding node. It chooses the most popular $\frac{x_i}{x_{i+1}}$ fraction of these objects and locally changes the replication level of the chosen objects to i, if their current replication level is $i+1$. The node also changes the replication level of the objects that are not chosen to $i+1$, if their current replication level is i or lower.

After the analysis phase, the replication level of some objects could increase or decrease, since the popularity of objects changes with time. If the replication level of an object decreases from level $i+1$ to i, it needs to be replicated in nodes that share one less prefix with it. If the replication level of an object increases from level i to $i+1$, the nodes with only i matching prefixes need to delete the replica. The *replicate phase* is responsible for enforcing the correct extent of replication for an object as determined by the analysis phase. During the replicate phase, each node A sends to each node B in the i^{th} level of its routing table, a *replication message* listing the identifiers of all objects for which B is the i level deciding node. When B receives this message from A, it checks the list of identifiers and pushes to node A any unlisted object whose current level of replication is i or lower. In addition, B sends back to A the identifiers of objects no longer replicated at level i. Upon receiving this message, A removes the listed objects.

Beehive nodes invoke the analysis and the replicate phases periodically. The analysis phase is invoked once every *analysis interval* and the replicate phase once every *replication interval*. In order to improve the efficiency of the replication protocol and reduce load on the network, we integrate the replication phase with the aggregation protocol. We perform this integration by setting the same durations for the replication interval and the aggregation interval and combining the replication and the aggregation messages as follows: When node A sends an aggregation message to B, the message contains the list of objects replicated at A whose i level deciding node is B. Similarly, when node B replies to the replication message from A, it adds the aggregated access frequency information for all objects listed in the replication message.

The analysis phase estimates the relative popularity of the objects using the estimates for access frequency obtained through the aggregation protocol. Recall that, for an object replicated at level i, it takes $2(logN - i)$ rounds of aggregation to obtain an accurate estimate of the access frequency. In order to allow time for the information flow during aggregation, we set the replication interval to at least $2logN$ times the aggregation interval.

Random variations in the query distribution will lead to fluctuations in the relative popularity estimates of objects, and may cause frequent changes in the replication levels of objects. This behavior may increase the object transfer activity and impose substantial load on the network. Increasing the duration of the aggregation interval is not an efficient solution because it decreases the responsiveness of system to changes. Beehive limits the impact of fluctuations by employing hysteresis. During the analysis phase, when a node sorts the objects at level i based on their popularity, the access frequencies of objects already replicated at level $i - 1$ is increased by a small fraction. This biases the system towards maintaining already existing replicas when the popularity difference between two objects is small.

The replication protocol also enables Beehive to maintain appropriate replication levels for objects when new nodes join and others leave the system. When a new node joins the system, it obtains the replicas of objects it needs to store by initiating a replicate phase of the replication protocol. If the new node already has objects replicated when it was previously part of the system, then these objects need not be fetched again from the deciding nodes. A node leaving the system does not directly affect Beehive. If the leaving node is a deciding node for some objects, the underlying DHT chooses a new deciding node for these objects when it repairs the routing table.

2.4 Mutable Objects

Beehive directly supports mutable objects by proactively disseminating object updates to the replicas in the system. The semantics of read and update operations on objects is an important issue to consider while supporting object mutability. Strong consistency semantics require that once an object is updated, all subsequent queries to that object only return the modified object. Achieving strong consistency is challenging in a distributed system with replicated objects, because each copy of the replicated object should be updated or invalidated upon object modification. In Beehive, we exploit the structure of the underlying DHT to efficiently disseminate object updates to all the nodes carrying replicas of the object. Our scheme guarantees that when an object is modified, all replicas will be consistently updated at all nodes.

Beehive associates a *version number* with each object to identify modified objects. An object replica with higher version number is more recent than a replica with lower version number. The owner of an object in the system can modify the object by inserting a fresh copy of the object with a higher version number at the home node. The home node proactively propagates the update to all the replicas of the objects using the routing table. If the object is replicated at level i, the home node sends a copy of the updated object to each node B in the i^{th} level of the routing table. Node B then propagates the update to each node in the $(i + 1)^{th}$ level of its routing table.

The update propagation protocol ensures that each node A sharing at least i prefixes with the object obtain a copy of the modified object. The object update reaches the node A following exactly the same path a query issued at the object's home node for node A's identifier would follow. Because of this property, all nodes with a replica of the object get exactly one copy of the modified object. Hence, this scheme is both efficient and provides guaranteed cache coherency in the absence of nodes leaving the system.

Nodes leaving the system may cause temporary inconsistencies in the routing table. Consequently, updates may not reach some nodes where objects are replicated. Beehive alleviates this problem by incorporating a lazy update propagation mechanism to the replicate phase. Each node includes in the replication message, the current version numbers of the replicated objects. Upon receiving this message, the deciding node pushes a copy of the object if it has a more recent version.

3 Implementation

Beehive is a general replication mechanism that can be applied to any prefix-based distributed hash table. We have layered our implementation on top of Pastry, a freely available DHT with log(N) lookup performance. Our implementation is structured as a transparent layer on top of FreePastry 1.3, supports a traditional insert/modify/delete/query DHT interface for applications, and required no modifications to underlying Pastry. However, converting the preceding discussion into a concrete implementation of the Beehive framework, building a DNS application on top, and combining the framework with Pastry required some practical considerations and identified some optimization opportunities.

Beehive needs to maintain some additional, modest amount of state in order to track the replication level, freshness, and popularity of objects. Each Beehive node stores all replicated objects in an object repository. Beehive associates the following meta-information with each object in the system, and each Beehive node maintains the following fields within each object in its repository:

- Object-ID: A 128-bit field uniquely identifies the object and helps resolve queries. The object identifier is derived from the hash key at the time of insertion, just as in Pastry.

- Version-ID: A 64-bit version number differentiates fresh copies of an object from older copies cached in the network.
- Home-Node: A single bit specifies whether the current node is the home node of the object.
- Replication-Level: A small integer specifies the current, local replication level of the object.
- Access-Frequency: An integer monitors the number of queries that have reached this node. It is incremented by one for each locally observed query, and reset at each aggregation.
- Aggregate-Popularity: A integer used in the aggregation phase to collect and sum up the access count from all dependent nodes for which this node is the home node. We also maintain an older aggregate popularity count for aging.

In addition to the state associated with each object, Beehive nodes also maintain a running estimate of the Zipf parameter. Overall, the storage cost consists of several bytes per object, and the processing cost of keeping the meta-data up to date is small.

Pastry's query routing deviates from the model described earlier in the paper because it is not entirely prefix-based and uniform. Since Pastry maps each object to the numerically closest node in the identifier space, it is possible for an object to not share any prefixes with its home node. For example, in a network with two nodes 298 and 315, Pastry will store an object with identifier 304 on node 298. Since a query for object 304 propagated by prefix matching alone cannot reach the home node, Pastry completes the query with the aid of an auxiliary data structure called *leaf set*. The leaf set is used in the last few hops to directly locate the numerically closest node to the queried object. Pastry initially routes a query using entries in the routing table, and may route the last couple of hops using the leaf set entries. This required us to modify Beehive's replication protocol to replicate objects at the leaf set nodes as follows. Since the leaf set is most likely to be used for the last hop, we replicate objects in the leaf set nodes only at the highest replication levels. Let $k = log_b N$ be the highest replication level for Beehive, that is, the default replication level for an object replicated only at its home node. As part of the replicate phase, a node A sends a replication message to all nodes B in its routing table as well as its leaf set with a list of identifiers of objects replicated at level $k-1$ whose deciding node is B. B is the deciding node of an object homed at node A, if A would forward a query to that object to node B next. Upon receiving a maintenance message at level $k-1$, node B would push an object to node A only if node A and the object have at least $k-1$ matching prefixes. Once an object is replicated on a leaf set node at level $k-1$, further replication to lower levels follow the replication protocol described in Section 2. This slight modification to Beehive enables it to work on top of Pastry. Other routing metrics for DHT substrates, such as the XOR metric [18], have been proposed that do not exhibit this non-uniformity, and where the Beehive implementation would be simpler.

Pastry's implementation provides two opportunities for optimization, which improve Beehive's impact and reduce its overhead. First, Pastry nodes preferentially populate their routing tables with nodes that are in physical proximity [4]. For instance, a node with identifier 100 has the opportunity to pick either of two nodes 200 and 201 when routing based on the first digit. Pastry selects the node with the lowest network latency, as measured by the packet round-trip time. As the prefixes get longer, node density drops and each node has progressively less freedom to find and choose between nearby nodes. This means that a significant fraction of the lookup latency experienced by a Pastry lookup is incurred on the last hop. Hence, selecting even a large number of constant hops, C, as Beehive's performance target, will have a significant effect on the real performance of the system. While we pick $C = 1$ in our implementation, note that C is a continuous variable and may be set to a fractional value, to get average lookup performance that is a fraction of a hop. $C = 0$ yields a solution that will replicate all objects at all hops, which is suitable only if the total hash table size is small.

The second optimization opportunity stems from the periodic maintenance messages used by Beehive and Pastry. Beehive requires periodic communication between nodes and the member of their routing table and leaf-set for replica dissemination and data aggregation. Pastry nodes periodically send heart-beat messages to nodes in their routing table and leaf set to detect node failures. They also perform periodic network latency measurements to nodes in their routing table in order to obtain closer routing table entries. We can improve Beehive's efficiency by combining the periodic heart-beat messages sent by Pastry with the periodic aggregation messages sent by Beehive. By piggy-backing the ith row routing table entries on to the Beehive aggregation message at replication level i, a single message can simultaneously serve as a heart beat message, Pastry maintenance message, and a Beehive aggregation message.

We have built a prototype DNS name server on top of Beehive in order to evaluate the caching strategy proposed in this paper. Beehive-DNS uses the Beehive framework to proactively disseminate DNS resource records containing name to IP address bindings. The Beehive-DNS server currently supports UDP-based queries and is compatible with widely-deployed resolver libraries. Queries that are not satisfied within the Beehive system are looked up in the legacy DNS by the home node and are inserted into the Beehive framework. The Beehive system stores and disseminates resource records to the appropriate replication levels by monitoring the DNS query stream. Clients are free to route their queries through any node that is part of the Beehive-DNS. Since the DNS system relies entirely on aggressive caching in order to scale, it provides very loose coherency semantics, and limits the rate at which updates can be performed. Recall that the Beehive system enables resource records to be modified at any time, and disseminates the new resource records to all caching name servers as part of the update operation. However, for this process to be initiated, name owners would have to directly notify the home node of changes to the name to IP address binding. We expect that, for some time to come, Beehive will be an adjunct system layered on top of legacy DNS, and therefore name owners who are not part of Beehive will not know to contact the system. For this reason, our current implementation delineates between names that exist solely in Beehive versus resource records originally inserted from legacy DNS. In the current implementation, the home node checks for the validity of each legacy DNS entry by issuing a DNS query for the domain when the time-to-live field of that entry is expired. If the DNS mapping has changed, the home node detects the update and propagates it as usual. Note that this strategy preserves DNS semantics and is quite efficient because only the home nodes check the validity of each entry, while replicas retain all mappings unless invalidated.

Overall, the Beehive implementation adds only a modest amount of overhead and complexity to peer-to-peer distributed hash tables. Our prototype implementation of Beehive-DNS is only 3500 lines of code, compared to the 17500 lines of code for Pastry.

4 Evaluation

In this section, we evaluate the performance costs and benefits of the Beehive replication framework. We examine Beehive's performance in the context of a DNS system and show that Beehive can robustly and efficiently achieve its targeted lookup performance. We also show that Beehive can adapt to sudden, drastic changes in the popularity of objects as well as global shifts in the parameter of the query distribution, and continue to provide good lookup performance.

We compare the performance of Beehive with that of pure Pastry and Pastry enhanced by passive caching. By passive caching, we mean caching objects along all nodes on the query path, similar to the scheme proposed in [23]. We impose no restrictions on the size of the cache used in passive caching. We follow the DNS cache model to handle mutable objects, and associate a time to live with each object. Objects are removed from the cache upon expiration of the time to live.

4.1 Setup

We evaluate Beehive using simulations, driven by a DNS survey and trace data. The simulations were performed using the same source code as our implementation. Each simulation run was started by seeding the network with just a single copy of each object, and then querying for objects according to a DNS trace. We compared the proactive replication of Beehive to passive caching in Pastry (PC-Pastry), as well as regular Pastry.

Since passive caching relies on expiration times for coherency, and since both Beehive and Pastry need to perform extra work in the presence of updates, we conducted a large-scale survey to determine the distribution of TTL values for DNS resource records and to compute the rate of change of DNS entries. Our survey spanned July through September 2003, and periodically queried web servers for the resource records of 594059 unique domain names, collected by crawling the Yahoo! and the DMOZ.ORG web directories. We used the distribution of the returned time-to-live values to determine the lifetimes of the resource records in our simulation. We measured the rate of change in DNS entries by repeating the DNS survey periodically, and derived an object lifetime distribution. We used this distribution to introduce a new version of an object at the home node.

We used the DNS trace [15] collected at MIT between 4 and 11 December 2000. This trace spans $4,160,954$ lookups over 7 days featuring 1233 distinct clients and $302,032$ distinct fully-qualified names. In order to reduce the memory consumption of the simulations, we scale the number of distant objects to 40960, and issue queries at the same rate of 7 queries per sec. The rate of issue for requests has little impact on the hit rate achieved by Beehive, which is dominated mostly by the performance of the analytical model, parameter estimation, and rate of updates. The overall query distribution of this trace follows an approximate Zipf-like distribution with parameter 0.91 [15]. We separately evaluate Beehive's robustness in the face of changes in this parameter.

We performed our evaluations by running the Beehive implementation on Pastry in simulator mode with 1024 nodes. For Pastry, we set the base to be 16, the leaf-set size to be 24, and the length of identifiers to be 128, as recommended in [22]. In all our evalua-

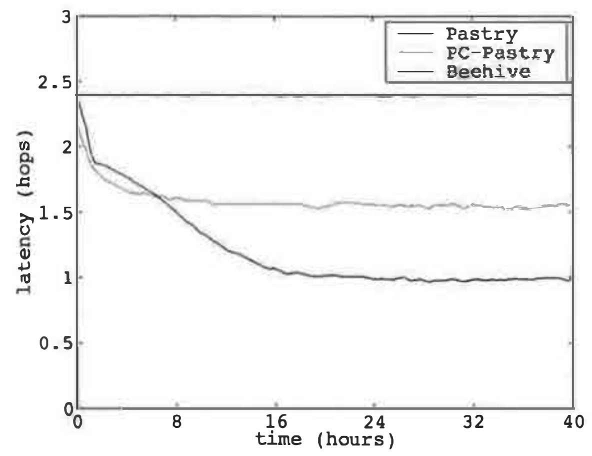


Figure 3: **Latency (hops) vs Time. The average lookup performance of Beehive converges to the targeted $C = 1$ hop after two replication phases.**

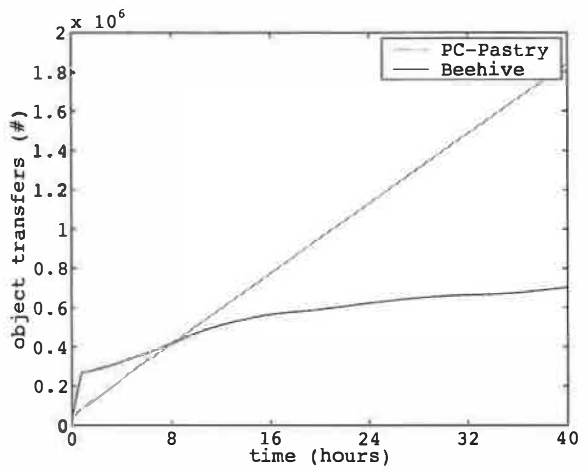


Figure 4: **Object Transfers (cumulative) vs Time. The total amount of object transfers imposed by Beehive is significantly lower compared to caching. Passive caching incurs large costs in order to check freshness of entries in the presence of conservative timeouts.**

tions, the Beehive aggregation and replication intervals were 48 minutes and the analysis interval was 480 minutes. The replication phases at each node were randomly staggered to approximate the behavior of independent, non-synchronized hosts. We set the target lookup performance of Beehive to average 1 hop.

Beehive Performance

Figure 3 shows the average lookup latency for Pastry, PC-Pastry, and Beehive over a query period spanning 40 hours. We plot the lookup latency as a moving average over 48 minutes. The average lookup latency of pure Pastry is about 2.34 hops. The average lookup latency of PC-Pastry drops steeply during the first 4 hours and averages 1.54 after 40 hours. The average lookup performance of Beehive decreases steadily and converges to about 0.98 hops, within 5% of the target lookup performance. Beehive achieves the target performance in about 16 hours and 48 minutes, the time required for two analysis phases followed by a replication phase at each node. These three phases, combined, enable Beehive to propagate the popular objects to their respective replication levels and achieve the expected payoff. In contrast, PC-Pastry provides limited benefits, despite an infinite-sized cache. There are two reasons for the relative ineffectiveness of passive caching. First, the heavy tail in Zipf-like distributions implies that there will be many objects for which there will be few requests, where queries will take many disjoint paths in the network until they collide on a node on which the object has been cached. Second, PC-Pastry relies on time-to-live values for cache coherency, instead of tracking the location of cached objects. The time-to-live values are set conservatively in order to reflect the worst case scenario under which the record may be updated, as opposed to the expected lifetime of the object. Our survey indicates that 95% of the DNS records have a lifetime of less than one day, whereas fewer than 0.8% of the records change in 24 hours. Consequently, passive caching suffers from a low hit rate as entries are evicted due to conservative values of TTL set by name owners.

Next, we examine the bandwidth consumed and the network load incurred by PC-Pastry and Beehive, and show that Beehive generates significantly lower background traffic due to object transfers compared to passive caching. Figure 4 shows the total amount of objects transferred by Beehive and PC-Pastry since the beginning of the experiment. PC-Pastry has an average object transfer rate proportional to its lookup latency, since it transfers an object to each node along the query path. Beehive incurs a high rate of object transfer during the initial period; but once Beehive achieves its target lookup performance, it incurs considerably lower overhead, as it needs to perform transfers only in response to changes in object popularity and, relatively infrequently for DNS, to object updates. Beehive continues to perform limited amounts of object replication, due to fluctuations in the popularity of the objects as well as estimation errors not dampened down by hysteresis. The rate of object transfers is initially high because the entire system is started at the same time with only one copy of each object. In practice, node-joins and object-inserts would be staggered in time allowing the system to operate smoothly without sudden spikes in bandwidth consumption.

The average number of objects stored at each node at the end of 40 hours is 380 for Beehive and 420 for pas-

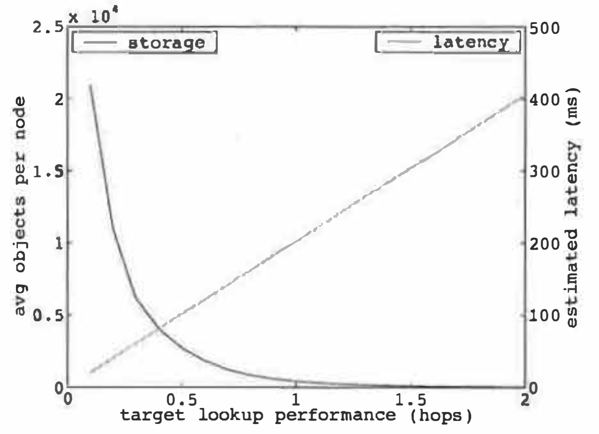


Figure 5: **Storage Requirement vs Latency. This graph shows the average per node storage required by Beehive and the estimated latency for different target lookup performance. This graph captures the trade off between the overhead incurred by Beehive and the lookup performance achieved.**

sive caching. PC-Pastry caches more objects than Beehive even though its lookup performance is worse, due to the heavy tailed nature of Zipf distributions. Beehive requires only 95 objects per node to provide 1.54 hops, the lookup performance achieved by PC-Pastry. Our evaluation shows that Beehive provides 1 hop average lookup latency with low storage and bandwidth overhead.

Beehive efficiently trades off storage and bandwidth for improved lookup latency. Our replication framework enables administrators to tune this trade off by varying the target lookup performance of the system. Figure 5 shows the trade off between storage requirement and estimated latency for different target lookup performance. We used the analytical model described in Section 2 to estimate the storage requirements. We estimated the expected lookup latency from round trip time obtained by pinging all pairs of nodes in PlanetLab, and adding to this 0.42 ms for accessing the local DNS resolver. The average 1 hop round trip time between nodes in PlanetLab is 202.2 ms (median 81.9 ms). In our large scale DNS survey, the average DNS lookup latency was 255.9 ms (median 112 ms). Beehive with a target performance of 1 hop can provide better lookup latency than DNS.

Flash Crowds

Next, we examine the performance of proactive and passive caching in response to changes in object popularity. We modify the trace to suddenly reverse the popularities of all the objects in the system. That is, the least popular object becomes the most popular object, the second least popular object becomes the second most popular object, and so on. This represents a worst case scenario for proactive replication, as objects that are least replicated suddenly need to be replicated widely, and vice versa. The switch occurs at $t = 40$, and we issue queries from the reversed popularity distribution for another 40 hours.

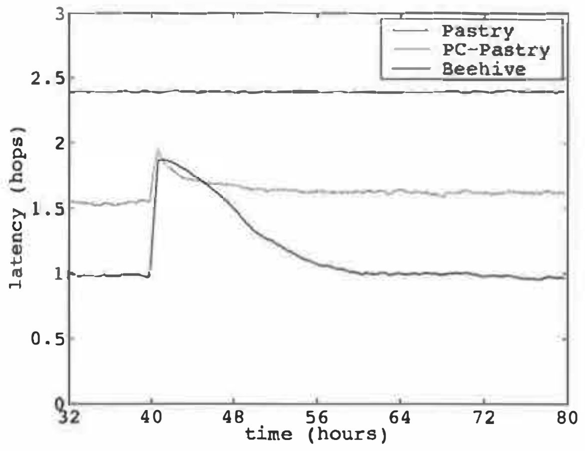


Figure 6: **Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the popularity of objects and brings the average lookup performance to one hop.**

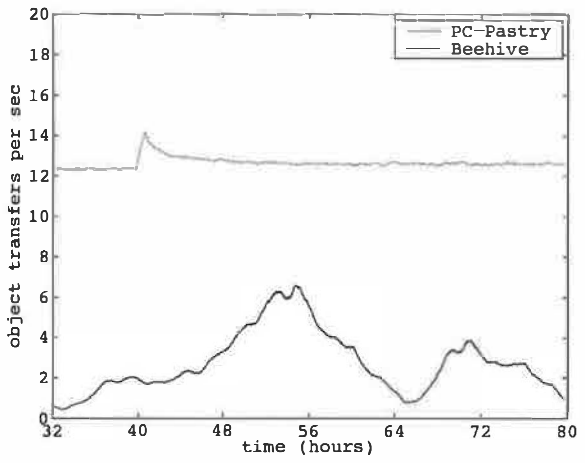


Figure 7: **Rate of Object Transfers vs Time. This graph shows that when popularity of the objects change, Beehive imposes extra bandwidth overhead temporarily to replicate the newly popular objects and maintain constant lookup time.**

Figure 6 shows the lookup performance of Pastry, PC-Pastry and Beehive in response to flash crowds. Popularity reversal causes a temporary increase in average latency for both Beehive and PC-Pastry. Beehive adjusts the replication levels of its objects appropriately and reduces the average lookup performance to about 1 hop after two replication intervals. The lookup performance of passive caching also decreases to about 1.6 hops. Fig-

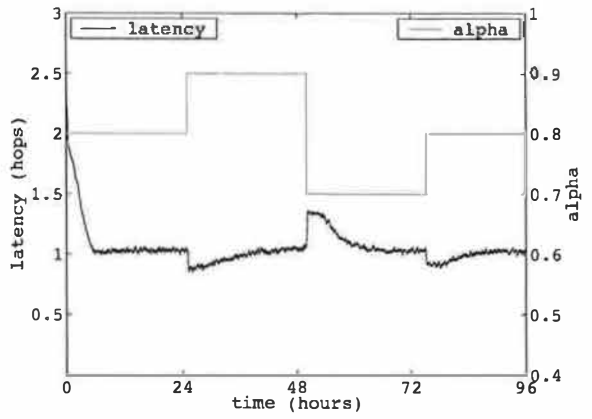


Figure 8: **Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the parameter of the query distribution and brings the average lookup performance to one hop.**

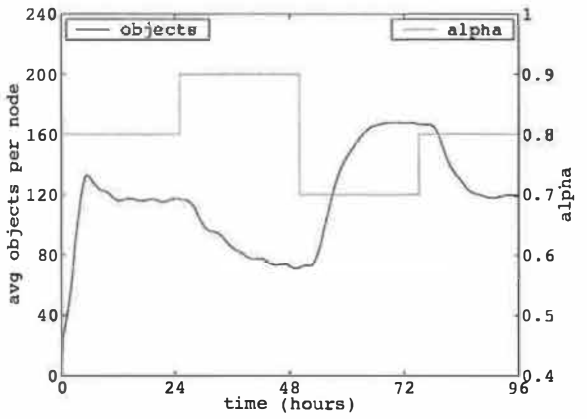


Figure 9: **Objects stored per node vs Time. This graph shows that when the parameter of the query distribution changes, Beehive adjusts the number of replicated objects to maintain O(1) lookup performance with storage efficiency.**

ure 7 shows the instantaneous rate of object transfer induced by the popularity reversal for Beehive and PC-Pastry. The popularity reversal causes a temporary increase in the object transfer activity of Beehive as it adjusts the replication levels of the objects appropriately. Even though Beehive incurs this high rate of activity in response to a worst-case scenario, it consumes less bandwidth and imposes less aggregate load compared to passive caching.

Zipf Parameter Change

Finally, we examine the adaptation of Beehive to global changes in the parameter of the overall query distribution. We issue queries from Zipf-like distributions generated with different values of the parameter α at each 24 hour interval. We seeded these simulations with 4096 objects.

Figure 8 shows the lookup performance of Beehive as it adapts to changes in the parameter of the query distribution. Beehive effectively detects changes in $alpha$ and redistributes object replicas to meet targeted performance objectives. Figure 9 shows the average number of objects replicated at each node in the system by Beehive. As α varies, so do the number of replicas Beehive creates on each node. Beehive increases the number of replicated objects per node as α decreases to meet the targeted performance goal, and reduces the number of replicas as $alpha$ increases to reclaim space. The number of replicas per node shown in Figure 9 agrees with optimal solution provided by the analytical model. Overall, continuously monitoring and estimating the α of the query distribution enables Beehive to adjust the extent and level of replication to compensate for drastic, global changes.

Summary

In this section, we have evaluated the performance of the Beehive replication framework for different scenarios in the context of DNS. Our evaluation indicates that Beehive achieves O(1) lookup performance with low storage and bandwidth overhead. In particular, it outperforms passive caching in terms of average latency, storage requirements, network load and bandwidth consumption. Beehive continuously monitors the popularity of the objects and the parameter of the query distribution, and quickly adapts its performance to changing conditions.

5 Related Work

Unstructured peer-to-peer systems, such as Freenet [5] and Gnutella [1] locate objects through on generic graph traversal algorithms, such as iterative depth-first search and flooding-based breadth-first search, respectively. These algorithms are inefficient, do not scale well, and do not provide sublinear bounds on lookup performance.

Several structured peer-to-peer systems, which provide a worst-case bound on lookup performance, have been proposed recently. CAN [21] maps both objects and nodes on a d-dimensional torus and provides O($dn^{\frac{1}{d}}$) lookup performance. Plaxton et al. [19] introduce a randomized lookup algorithm based on prefix matching to locate objects in a distributed network in O($logN$) probabilistic time. Chord [24], Pastry [22], and Tapestry [28] use consistent hashing to map objects to nodes and use Plaxton's prefix-matching algorithms to achieve O($logN$) worst-case lookup performance. Kademlia [24] also provides O($logN$) lookup performance using a similar search technique, but uses the XOR metric for routing. Viceroy [17] provides O($logN$) lookup performance with a constant degree routing graph. De Bruijn graphs [16, 26] provide O($logN$) lookup performance with 2 neighbors per node and O($logN/loglogN$) with $logN$ degree per node. Beehive can be applied to any of the overlays based on prefix-matching.

A few recently introduced DHTs provide O(1) lookup performance. Kelips [12] probabilistically provides O(1) lookup performance by dividing the network into O($\sqrt{N}$) affinity groups of O($\sqrt{N}$) nodes, replicating every object on every node within an affinity group, and using gossip to propagate updates. An alternative method [13] relies on maintaining full routing state, that is, a complete list of all system members, at each node. Farsite [10] uses routing tables of size O($dn^{\frac{1}{d}}$) to route in O(d) hops, but does not address rapid membership changes. Beehive fundamentally differs from these systems in three fundamental ways. First, Beehive can serve queries in less than one hop on average. And second, it achieves low storage overhead, bandwidth consumption and network load by minimizing the amount of replicated data in the system. Finally, Beehive provides a fine grain control of the trade off between lookup performance and overhead by allowing users to choose the target lookup performance from a continuous range.

Some DHTs pick routing table entries based on network proximity. Recent work [4, 27] has shown that this method can reduce the total lookup latency to a constant. However, the performance improvement is limited in large networks because the achieved absolute latency is several times more than the average single-hop latency.

Several peer-to-peer applications, such as PAST [23] and CFS [9], incorporate caching and replication. Both reserve a part of the storage space at each node to cache query results on the lookup path in order to improve subsequent queries. They also maintain a constant number of replicas of each object in the system in order to improve fault tolerance. As shown in this paper, passive caching schemes achieve limited improvement and do not provide provide performance guarantees.

Some systems employ a combination of caching with proactive object updates. In [6], the authors describe a proactive cache for DNS records where the cache proactively refreshes DNS records as they expire. While this technique reduces the impact of short expiration times on lookup performance, it introduces a large amount of overhead and does not qualitatively improve lookup performance. Controlled Update Propagation (CUP) [20] is a demand-based caching mechanism with proactive object updates. CUP nodes propagate object updates away from a designated home node in accordance to a popularity based *incentive* that flows from the leaf nodes towards the home node. While there are some similarities between the replication protocols of CUP and Beehive, the decision to cache objects and propagate updates in CUP are based on heuristics. [25] describes a distributed hierarchical web cache that replicates objects proactively, selects replica locations based on heuristics and pushes updates to replicas.

The closest work to Beehive is [7], which examines optimal strategies for replicating objects in unstructured peer-to-peer systems. This paper analytically derives the optimal number of randomly-placed object replicas in unstructured peer-to-peer systems. The observations in this work are not directly applicable to structured DHTs, because it assumes that the lookup time for an object depends only on the number of replicas and not the placement strategy. Beehive achieves higher performance with fewer replicas by exploiting the structure of the underlying overlay.

6 Future Work

This paper has investigated the potential performance benefits of model-driven proactive caching and has shown that it is feasible to use peer-to-peer systems in cooperative low-latency, high-performance environments. Deploying full-blown applications, such as a complete peer-to-peer DNS replacement, on top of this substrate will require substantial further effort. Most notably, security issues need to be addressed before peer-to-peer systems can be deployed widely. At the application level, this involves using some authentication technique, such as DNSSEC [11], to securely delegate name service to nodes in a peer to peer system. At the underlying DHT layer, secure routing techniques [3] are required to limit the impact of malicious nodes on the DHT. Both of these techniques will add additional latencies, which may be offset at the cost of additional bandwidth, storage and load by setting Beehive's target performance to lower values. At the Beehive layer, the proactive replication layer needs to be protected from nodes that misreport the popularity of objects. Since a malicious peer in Beehive can replicate an object, or indirectly cause an object to be replicated, at b nodes that have that malicious node in their routing tables, we expect that one can limit the amount of damage that attackers can cause through misreported object popularities.

7 Conclusion

Structured DHTs offer many desirable properties for a large class of applications, including self-organization, failure resilience, high scalability, and a worst-case performance bound. However, their O($logN$) hop average-case performance has prohibited them from serving latency-sensitive applications.

In this paper, we outline a framework for proactive replication that offers O(1) DHT lookup performance for a frequently encountered class of query distributions. At the core of this framework is an analytical model that yields the optimal object replication required to achieve constant time lookups. Beehive achieves high performance by decoupling lookup performance from the size of the network. It adapts quickly to flash crowds, detects changes in the global query distribution and self-adjusts to retain its performance guarantees. Overall, Beehive enables DHTs to be used for serving latency-sensitive applications.

Acknowledgments

We are grateful to Robert Morris, Hari Balakrishnan, Jaeyon Jung, and Emil Sit for providing us with their DNS traces.

References

[1] "The Gnutella Protocol Specification v.0.4." *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*, Mar 2001.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications." *IEEE INFOCOM 1999*, New York NY, Mar 1999.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. "Secure Routing for Structured Peer-to-Peer Overlay Networks." *OSDI 2002*, Boston MA, Dec 2002.

[4] M. Castro, P. Druschel, C. Hu, and A. Rowstron. "Exploiting Network Proximity in Peer-to-Peer Overlay Networks." *Technical Report MSR-TR-2002-82, Microsoft Research*, May 2002.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." *Lecture Notes in Computer Science*, 2009:46-66, 2001.

[6] E. Cohen and H. Kaplan. "Proactive Caching of DNS Records: Addressing a Performance Bottleneck." *SAINT 2001*, San Diego CA, Jan 2001.

[7] E. Cohen and S. Shenker. "Replication Strategies in Unstructured Peer-to-Peer Networks." *ACM SIGCOMM 2002*, Pittsburgh PA, Aug 2002.

[8] R. Cox, A. Muthitacharoen, and R. Morris. "Serving DNS using a Peer-to-Peer Lookup Service." *IPTPS 2002*, Cambridge MA, Mar 2002.

[9] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. "Wide-Area Cooperative Storage with CFS." *ACM SOSP 2001*, Banff Alberta, Canada, Oct 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. "Reclaiming Space from Duplicate Files in a Serverless Distributed File System." *ICDCS 2002*, Vienna, Austria, Jul 2002.

[11] D. Eastlake. "Domain Name System Security Extensions". *Request for Comments (RFC) 2535*, 3^{rd} edition, Mar 1999.

[12] I. Gupta, K. Birman, P. Linga, A. Demers, and R. v. Renesse. "Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead." *IPTPS 2003*, Berkeley CA, Feb 2003.

[13] A. Gupta, B. Liskov, R. Rodrigues. "One Hop Lookups for Peer-to-Peer Overlays." *HotOS 2003*. Lihue HI, May 2003.

[14] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. "SkipNet: A Scalable Overlay Network with Practical Locality Properties.", *USITS 2003*, Seattle WA, Mar 2003.

[15] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. "DNS Performance and Effectiveness of Caching." *ACM SIGCOMM Internet Measurement Workshop 2001*, San Francisco CA, Nov 2001.

[16] F. Kaashoek and D. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table." *IPTPS 2003*, Berkeley CA, Feb 2003.

[17] D. Malkhi, M. Naor, and D. Ratajczak. "Viceroy: A Scalable and Dynamic Emulation of the Butterfly" *ACM PODC 2002*, Monterey CA, Aug 2002.

[18] P. Maymounkov and D. Maziéres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric." *IPTPS 2002*, Cambridge MA, Mar 2002.

[19] G. Plaxton, R. Rajaraman, and A. Richa. "Accessing nearby copies of replicated objects in a distributed environment." *Theory of Computing Systems*, 32:241-280, 1999.

[20] M. Roussopoulos and M. Baker. "CUP: Controlled Update Propagation in Peer-to-Peer Networks." *USENIX 2003 Annual Technical Conference*, San Antonio TX, Jun 2003.

[21] S. Ratnasamy, P. Francis, M. Hadley, R. Karp, and S. Shenker. "A Scalable Content-Addressable Network." *ACM SIGCOMM 2001*, San Diego CA, Aug 2001.

[22] A. Rowstorn and P. Druschel. "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems." *IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov 2001.

[23] A. Rowstorn and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale Persistent Peer-to-Peer Storage Utility." *ACM SOSP 2001*, Banff Alberta, Canada, Oct 2001.

[24] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." *ACM SIGCOMM 2001*, San Diego CA, Aug 2001.

[25] R. Tewari, M. Dahlin, H. Vin, and J. Kay. "Design Considerations for Distributed Caching on the Internet." *ICDCS 1999*, Austin TX, Jun 1999.

[26] U. Wieder and M. Naor. "A Simple Fault Tolerant Distributed Hash Table." *IPTPS 2003*, Berkeley CA, Feb 2003.

[27] H. Zhang, A. Goel, and R. Govindan. "Incrementally Improving Lookup Latency in Distributed Hash Table Systems." *SIGMETRICS 2003*, San Diego CA, Jun 2003.

[28] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. "Tapestry: A Resilient Global-scale Overlay for Service Deployment." *IEEE Journal on Selected Areas in Communications, JSAC*, 2003.

Efficient Routing for Peer-to-Peer Overlays

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues
MIT Computer Science and Artificial Intelligence Laboratory
{anjali,liskov,rodrigo}@csail.mit.edu

Abstract

Most current peer-to-peer lookup schemes keep a small amount of routing state per node, typically logarithmic in the number of overlay nodes. This design assumes that routing information at each member node must be kept small, so that the bookkeeping required to respond to system membership changes is also small, given that aggressive membership dynamics are expected. As a consequence, lookups have high latency as each lookup requires contacting several nodes in sequence.

In this paper, we question these assumptions by presenting two peer-to-peer routing algorithms with small lookup paths. First, we present a one-hop routing scheme. We show how to disseminate information about membership changes quickly enough so that nodes maintain accurate routing tables with complete membership information. We also deduce analytic bandwidth requirements for our scheme that demonstrate its feasibility.

We also propose a two-hop routing scheme for large scale systems of more than a few million nodes, where the bandwidth requirements of one-hop routing can become too large. This scheme keeps a fixed fraction of the total routing state on each node, chosen such that the first hop has low latency, and thus the additional delay is small.

We validate our analytic model using simulation results that show that our algorithms can maintain routing information sufficiently up-to-date such that a large fraction (e.g., 99%) of the queries will succeed without being re-routed.

1 Introduction

Structured peer-to-peer overlays like Chord [15], CAN [11], Pastry [13], and Tapestry [18] provide a substrate for building large-scale distributed applications. These overlays allow applications to locate objects stored in the system in a limited number of overlay hops.

Peer-to-peer lookup algorithms strive to maintain a small amount of per-node routing state – typically $O(\log N)$ – because their designers expect that system membership changes frequently. This expectation has been confirmed for successfully deployed systems. A recent study [14] shows that the average session time in Gnutella is only 2.9 hours. This is equivalent to saying that in a system with 100,000 nodes, there are about 19 membership change events per second.

Maintaining small tables helps keep the amount of bookkeeping required to deal with membership changes small. However, there is a price to pay for having only a small amount of routing state per node: lookups have high latency since each lookup requires contacting several nodes in sequence.

This paper questions the need to keep routing state small. We take the position that maintaining full routing state (i.e., a complete description of system membership) is viable even in a very large system, e.g., containing a million nodes. We present techniques that show that in systems of this size, nodes can maintain membership information accurately yet the communication costs are low. The results imply that a peer-to-peer system can route very efficiently even though the system is large and membership is changing rapidly.

We present a novel peer-to-peer lookup system that maintains complete membership information at each node. We show analytic results that prove that the system meets our goals of reasonable accuracy and bandwidth usage. It is, of course, easy to achieve these goals for small systems. Our algorithm is designed to scale to large systems. Our analysis shows that we can use one-hop routing for systems of up to a few millions of nodes.

Our analysis also shows that beyond a few million nodes, the bandwidth requirements of the one-hop scheme become too large. We present the design of a two-hop lookup scheme that overcomes this problem, and still provides faster lookups than existing peer-to-peer routing algorithms. We also present an analytic model of the two-hop system and conclude that its bandwidth requirements are reasonable, even for systems with tens of millions of nodes.

Finally, the paper presents simulation results that corroborate what our analytic models predict. We also show that performance does not degrade significantly as the system becomes larger or smaller than due to aggressive system dynamics.

The rest of the paper is organized as follows. Section 2 presents our system model. Sections 3 and 4 describe our one-hop and two-hop routing schemes, respectively. Section 5 evaluates our system. We conclude with a discussion of what we have accomplished.

2 System Model

We consider a system of n nodes, where n is a large number like 10^5 or 10^6. We assume dynamic membership behavior as in Gnutella, which is representative of an open Internet environment. From the study of Gnutella and Napster [14], we deduce that systems of 10^5 and 10^6 nodes would show around 20 and 200 membership changes per second, respectively. We call this rate r. We refer to membership changes as events in the rest of the paper.

Every node in the overlay is assigned a random 128-bit node identifier. Identifiers are ordered in an *identifier ring* modulo 2^{128}. We assume that identifiers are generated such that the resulting set is uniformly distributed in the identifier space, for example, by setting a node's identifier to be the cryptographic hash of its network address. Every node has a predecessor and a successor in the identifier ring, and it periodically sends keep-alive messages to these nodes.

Similarly, each item has a *key*, which is also an identifier in the ring. Responsibility for an item (e.g., providing storage for it) rests with its *successor*; this is the first node in the identifier ring clockwise from *key*. This mapping from keys to nodes is based on the one used in Chord [15], but changing our system to use other mappings is straightforward.

Clients issue queries that try to reach the successor node of a particular identifier. We intend our system to satisfy a large fraction, f, of the queries correctly on the *first* attempt (where each attempt requires one or two hops, depending on which scheme we use). Our goal is to support high values of f, e.g., $f = 0.99$. A query may fail in its first attempt due to a membership change, if the notification of the change has not reached the querying node. In such a case, the query can still be rerouted and succeed in a higher number of hops. Nevertheless, we define failed queries as those that are not answered correctly in the *first* attempt, as our objective is to have one- or two-hop lookups, depending on which algorithm we use.

3 One Hop Lookups

This section presents the design and analysis of our one-hop scheme. In this scheme, every node maintains a full routing table containing information about every other node in the overlay. The actual query success rate depends on the accuracy of this information.

Section 3.1 describes how the algorithm handles membership changes, namely how to convey information about these changes to all the nodes in the ring. Section 3.2 explains how the algorithm reacts to node failures and presents an informal correctness argument for our approach. Section 3.3 discusses issues about asymmetry in the load of individual nodes. Section 3.4 presents an analysis of the bandwidth requirements of this scheme.

3.1 Membership Changes

Membership changes (i.e., nodes joining and leaving the ring) raise two important issues that our algorithm must address. First, we must update local information about the membership change, in order for each node in the system to determine precisely which interval in the id space it is responsible for. The second issue is conveying information about the change to all the nodes in the ring so that these nodes will maintain correct information about the system membership and consequently manage to route in a single hop.

To maintain correct local information (i.e., information about each node's successor and predecessor node), every node n runs a stabilization routine periodically, wherein it sends keep-alive messages to its successor s and predecessor p. Node s checks if n is indeed its predecessor, and if not, it notifies n of the existence of another node between them. Similarly p checks if n is indeed its successor, and if not it notifies n. If either of s or p does not respond, n pings it repeatedly until a time-out period when it decides that the node is unreachable or dead.

A joining node contacts another system node to get its view of the current membership; this protocol is similar to the Chord protocol [15, 16]. The membership information enables it to get in touch with its predecessor and successor, thus informing them of its presence.

To maintain correct full routing tables, notifications of membership change events, i.e., joins and leaves, must reach every node in the system within a specified amount of time (depending on what frac-

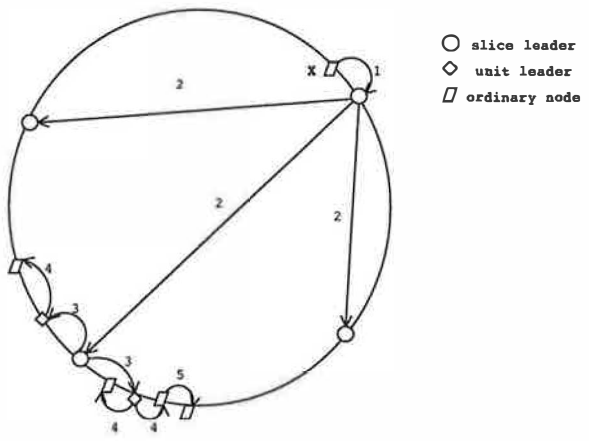


Figure 1. Flow of event notifications in the system

tion of failed queries, i.e., f, is deemed acceptable). Our goal is to do this in a way that has low notification delay yet reasonable bandwidth consumption, since bandwidth is likely to be the scarcest resource in the system.

We achieve this goal by superimposing a well-defined hierarchy on the system. This hierarchy is used to form dissemination trees, which are used to propagate event information.

We impose this hierarchy on a system with dynamic membership by dividing the 128-bit circular identifier space into k equal contiguous intervals called *slices*. The ith slice contains all nodes currently in the overlay whose node identifiers lie in the range $[i \cdot 2^{128}/k, (i+1) \cdot 2^{128}/k)$. Since nodes have uniformly distributed random identifiers, these slices will have about the same number of nodes at any time. Each slice has a *slice leader*, which is chosen dynamically as the node that is the successor of the mid-point of the slice identifier space. For example, the slice leader of the ith slice is the successor node of the key $(i + 1/2) \cdot 2^{128}/k$. When a new node joins the system it learns about the slice leader from one of its neighbors along with other information like the data it is responsible for and its routing table.

Similarly, each slice is divided into equal-sized intervals called *units*. Each unit has a *unit leader*, which is dynamically chosen as the successor of the mid-point of the unit identifier space.

Figure 1 depicts how information flows in the system. When a node (labeled **X** in Figure 1) detects a change in membership (its successor failed or it has a new successor), it sends an event notification message to its slice leader (**1**). The slice leader collects all event notifications it receives from its own slice and aggregates them for t_{big} seconds before sending a message to other slice leaders (**2**). To spread out bandwidth utilization, communication with different slice leaders is not synchronized: the slice leader ensures only that it communicates with each individual slice leader once every t_{big} seconds. Therefore, messages to different slice leaders are sent at different points in time and contain different sets of events.

The slice leaders aggregate messages they receive for a short time period t_{wait} and then dispatch the aggregate message to all unit leaders of their respective slices (**3**). A unit leader piggybacks this information on its keep-alive messages to its successor and predecessor (**4**).

Other nodes propagate this information in one direction: if they receive information from their predecessors, they send it to their successors and vice versa. The information is piggy-backed on keep-alive messages. In this way, all nodes in the system receive notification of all events, but within a unit information is always flowing from the unit leader to the ends of the unit. Nodes at unit boundaries do not send information to their neighboring nodes outside their unit. As a result, there is no redundancy in the communications: a node will get information only from its neighbor that is one step closer to its unit leader.

We get several benefits from choosing this design. First, it imposes a structure on the system, with well-defined event dissemination trees. This structure helps us ensure that there is no redundancy in communications, which leads to efficient bandwidth usage.

Second, aggregation of several events into one message allows us to avoid small messages. Small messages are a problem since the protocol overhead becomes significant relative to the message size, leading to higher bandwidth usage. This effect will be analyzed in more detail in Section 3.4.

Our scheme is a three-level hierarchy. The choice of the number of levels in the hierarchy involves a tradeoff: A large number of levels implies a larger delay in propagating the information, whereas a small number of levels generates a large load at the nodes in the upper levels. We chose a three level hierarchy because it has low delay, yet bandwidth consumption at top level nodes is reasonable.

3.2 Fault Tolerance

If a query fails on its first attempt it does not return an error to an application. Instead, queries can be rerouted. If a lookup query from node n_1 to node n_2 fails because n_2 is no longer in the system, n_1 can retry the query by sending it to n_2's successor. If the query failed because a recently joined node, n_3, is the new successor for the key that n_1 is looking up, n_2 can reply with the identity of n_3 (if it knows about n_3), and n_1 can contact n_3 in a second routing step.

Since our scheme is dependent on the correct functioning of slice leaders, we need to recover from their failure. Since there are relatively few slice leaders, their failures are infrequent. Therefore, we do not have to be very aggressive about replacing them in order to maintain our query success target. When a slice or unit leader fails, its successor soon detects the failure and becomes the new leader.

Between the time a slice or unit leader fails, and a new node takes over, some event notification messages may be lost, and the information about those membership changes will not be reflected in the system nodes' membership tables. This is not an issue for routing correctness, since each node maintains correct information about its predecessor and successor. It will, however, lead to more routing hops and if we allowed these errors to accumulate, it would eventually lead to a degradation of the one hop lookup success rate.

To avoid this accumulation, we use the lookups themselves to detect and propagate these inaccuracies. When a node performs a lookup and detects that its routing entry is incorrect (i.e., the lookup timed out, or was re-routed to a new successor), this new information is then pushed to all the system nodes via the normal channels: it notifies its slice leader about the event.

The correctness of our protocols is based on the fact that successor and predecessor pointers are correct. This ensures that, even if the remainder of the membership information contains errors, the query will eventually succeed after re-routing. In other words, our complete membership description can be seen as an optimization to following successor pointers, in the same way as Chord fingers are an optimization to successors (or similarly for other peer-to-peer routing schemes). Furthermore, we can argue that our successor and predecessor pointers are correct due to the fact that we essentially follow the same protocol as Chord to maintain these, and this has already been proven correct [16].

3.3 Scalability

Slice leaders have more work to do than other nodes, and this might be a problem for a poorly provisioned node with a low bandwidth connection to the Internet. To overcome this problem we can identify well connected and well provisioned nodes as "supernodes" on entry into the system (as in [17]). There can be a parallel ring of supernodes, and the successor (in the supernode ring) of the midpoint of the slice identifier space becomes the slice leader. We do require a sufficient number of supernodes to ensure that there are at least a few per slice.

As we will show in Section 3.4, bandwidth requirements are small enough to make most participants in the system potential supernodes in a 10^5 sized system (in such a system, slice leaders will require 35 kbps upstream bandwidth). In a million-node system we may require supernodes to be well-connected academic or corporate users (the bandwidth requirements increase to 350 kbps). Section 4 presents the two-hop scheme that may be required when we wish the system to accommodate even larger memberships.

3.4 Analysis

This section presents an analysis of how to parameterize the system to satisfy our goal of fast propagation. To achieve our desired success rate, we need to propagate information about events within some time period t_{tot}; we begin this section by showing how to compute this quantity. Yet we also require good performance, especially with respect to bandwidth utilization. Later in the section we show how we satisfy this requirement by controlling the number of slices and units.

Our analysis considers only non-failure situations. It does not take into account overheads of slice and unit leader failure because these events are rare. It also ignores message loss and delay since this simplifies the presentation, and the overhead introduced by message delays and retransmissions is small compared to other costs in the system.

Our analysis assumes that query targets are distributed uniformly throughout the ring. It is based on a worst case pattern of events, queries, and notifications: we assume all events happen just after the last slice-leader notifications, and all queries happen immediately after that, so that none of the affected routing table entries has been corrected and *all* queries targeted at those nodes (i.e., the nodes causing the events) fail. In a real deployment, queries would be interleaved with events and notifications, so fewer of them would fail.

This scenario is illustrated by the timeline in Figure 2. Here t_{wait} is the frequency with which slice leaders communicate with their unit leaders, t_{small} is the time it takes to propagate information throughout a unit, and t_{big} is the time a slice leader

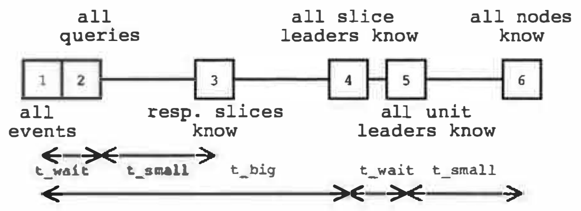


Figure 2. Timeline of the worst case situation

waits between communications to some other slice leader. Within $t_{wait}+t_{small}$ seconds (point 3), slices in which the events occurred all have correct entries for nodes affected by the respective events. After t_{big} seconds of the events (point 4), slice leaders notify other slice leaders. Within a further $t_{wait} + t_{small}$ seconds (point 6), all nodes in the system receive notification about all events.

Thus, $t_{tot} = t_{detect} + t_{wait} + t_{small} + t_{big}$. The quantity t_{detect} represents the delay between the time an event occurs and when the leader of that slice first learns about it.

3.4.1 Configuration Parameters

The following parameters characterize a system deployment:

1. f is the acceptable fraction of queries that fail in the first routing attempt
2. n is the expected number of nodes in the system
3. r is the expected rate of membership changes in the system

Given these parameters, we can compute t_{tot}. Our assumption that query targets are distributed uniformly around the ring implies that the fraction of failed queries is proportional to the expected number of incorrect entries in a querying node's routing table. Given our worst case assumption, all the entries concerning events that occurred in the last t_{tot} seconds are incorrect and therefore the fraction of failed queries is $\frac{r \times t_{tot}}{n}$. Therefore, to ensure that no more than a fraction f of queries fail we need:

$$t_{tot} \leq \frac{f \times n}{r}$$

For a system with 10^6 nodes, with a rate of 200 events/s, and $f = 1\%$, we get a time interval as large as $50s$ to propagate all information. Note also that if r is linearly proportional to n, then t_{tot} is independent of n. It is only a function of the desired success rate.

3.4.2 Slices and Units

Our system performance depends on the number of slices and units:

1. k is the number of slices the ring is divided into.
2. u is the number of units in a slice.

Parameters k and u determine the expected unit size. This in turn determines t_{small}, the time it takes for information to propagate from a unit leader to all members of a unit, given an assumption about h, the frequency of keep-alive probes. From t_{small} we can determine t_{big} from our calculated value for t_{tot}, given choices of values for t_{wait} and t_{detect}. (Recall that $t_{tot} = t_{detect} + t_{big} + t_{wait} + t_{small}$.)

To simplify the analysis we will choose values for h, t_{detect}, and t_{wait}. As a result our analysis will be concerned with just two independent variables, k and u, given a particular choice of values for n, r, and f. We will use one second for both h and t_{wait}. This is a reasonable decision since the amount of data being sent in probes and messages to unit leaders is large enough to make the overhead in these messages small (e.g., information about 20 events will be sent in a system with 10^5 nodes). Note that with this choice of h, t_{small} will be half the unit size. We will use three seconds for t_{detect} to account for the delay in detecting a missed keep-alive message and a few probes to confirm the event.

3.4.3 Cost Analysis

Our goal is to choose values for k and u in a way that reduces bandwidth utilization. In particular we are concerned with minimizing bandwidth use at the slice leaders, since they have the most work to do in our approach.

Bandwidth is consumed both to propagate the actual data, and because of the message overhead. m bytes will be required to describe an event, and the overhead per message will be v.

There are four types of communication in our system.

1. *Keep-alive messages:* Keep-alive messages form the base level communication between a node and its predecessor and successor. These messages include information about recent events. As described in Section 3.1, our system avoids sending redundant information in these messages by controlling the direction of information flow (from unit leader to unit members) and by not sending information across unit boundaries.

	Upstream	Downstream
Slice Leader	$r \cdot m \cdot (u+2) + \frac{2 \cdot v \cdot k}{t_{big}}$	$r \cdot m + \frac{2 \cdot v \cdot k}{t_{big}}$
Unit Leader	$2 \cdot r \cdot m + 3 \cdot v$	$r \cdot m + 2 \cdot v$
Other nodes	$r \cdot m + 2 \cdot v$	$r \cdot m + 2 \cdot v$

Table 1. Summary of bandwidth use

Since keep-alive messages are sent every second, every node that is not on the edge of a unit will send and acknowledge an aggregate message containing, on average, r events. The size of this message is therefore $r \cdot m + v$ and the size of the acknowledgment is v.

2. *Event notification to slice leaders:* Whenever a node detects an event, it sends a notification to its slice leader. The expected number of events per second in a slice is $\frac{r}{k}$. The downstream bandwidth utilization on slice leaders is therefore $\frac{r \cdot (m+v)}{k}$. Since each message must be acknowledged, the upstream utilization is $\frac{r \cdot v}{k}$.

3. *Messages exchanged between slice leaders:* Each message sent from one slice leader to another batches together events that occurred in the last t_{big} seconds in the slice. The typical message size is, therefore, $\frac{r}{k} \cdot t_{big} \cdot m + v$ bytes. During any t_{big} period, a slice leader sends this message to all other slice leaders ($k - 1$ of them), and receives an acknowledgment from each of them. Since each slice leader receives as much as it gets on average, the upstream and downstream use of bandwidth is symmetric. Therefore, the bandwidth utilization (both upstream and downstream) is

$$\left(\frac{r \cdot m}{k} + \frac{2 \cdot v}{t_{big}}\right) \cdot (k-1)$$

4. *Messages from slice leaders to unit leaders:* Messages received by a slice leader are batched for one second and then forwarded to unit leaders. In one second, r events happen and therefore the aggregate message size is $(r \cdot m + v)$ and the bandwidth utilization is

$$(r \cdot m + v) \cdot u$$

Table 1 summarizes the net bandwidth use on each node. To clarify the presentation, we have removed insignificant terms from the expressions.

Using these formulas we can compute the load on non-slice leaders in a particular configuration. In this computations we use $m = 10$ bytes and $v = 20$ bytes. In a system with 10^5 nodes, we see that the

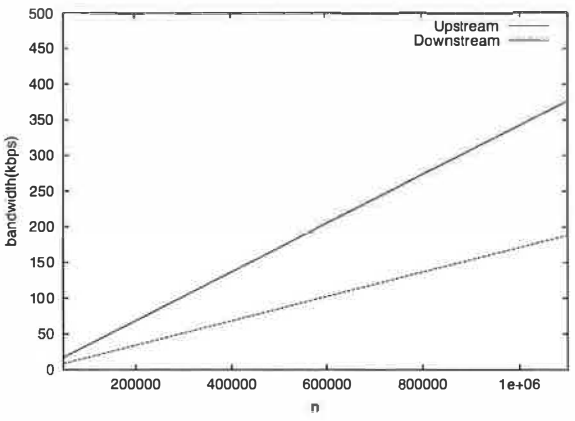


Figure 3. Bandwidth use on a slice leader with $r \propto n$

load on an ordinary node is 3.84 kbps and the load on a unit leader is 7.36 kbps upstream and 3.84 kbps downstream. For a system with 10^6 nodes, these numbers become 38.4 kbps, 73.6 kbps, and 38.4 kbps respectively.

From the table it is clear that the upstream bandwidth required for a slice leader is likely to be the dominating and limiting term. Therefore, we shall choose parameters that minimize this bandwidth. By simplifying the expression and using the interrelationship between u and t_{big} (explained in Section 3.4.2) we get a function that depends on two independent variables k and u. By analyzing the function, we deduce that the minimum is achieved for the following values:

$$k = \sqrt{\frac{r \cdot m \cdot n}{4 \cdot v}}$$

$$u = \sqrt{\frac{4 \cdot v \cdot n}{r \cdot m \cdot (t_{tot} - t_{wait} - t_{detect})^2}}$$

These formulas allow us to compute values for k and u. For example in a system of 10^5 nodes we want roughly 500 slices each containing 5 units. In a system of 10^6 nodes, we still have 5 units per slice, but now there are 5000 slices.

Given values for k and u we can compute the unit size and this in turn allows us to compute t_{small} and t_{big}. We find that we use least bandwidth when

$$t_{small} = t_{big}$$

Thus, we choose 23 seconds for t_{big} and 23 seconds for t_{small}.

Given these values and the formulas in Table 1, we can plot the bandwidth usage per slice leader in

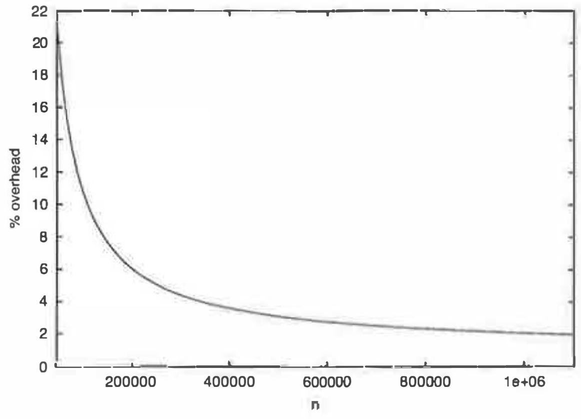


Figure 4. Aggregate bandwidth overhead of the scheme as a percentage of the theoretical optimum

systems of various sizes. The results of this calculation are shown in Figure 3. Note that the load increases only linearly with the size of the system. The load is quite modest in a system with 10^5 nodes (35 kbps upstream bandwidth), and therefore even nodes behind cable modems can act as slice leaders in such a system. In a system with 10^6 nodes the upstream bandwidth required at a slice leader is approximately 350 kbps. Here it would be more appropriate to limit slice leaders to being machines on reasonably provisioned local area networks. For larger networks, the bandwidth increases to a point where a slice leader would need to be a well-provisioned node.

Figure 4 shows the percentage overhead of this scheme in terms of aggregate bandwidth used in the system with respect to the hypothetical optimum scheme with zero overhead. In such a scheme scheme, the cost is just the total bandwidth used in sending r events to every node in the system every second, i.e., $r \cdot n \cdot m$. Note that the overhead in our system comes from the per-message protocol overhead. The scheme itself does not propagate any redundant information. We note that the overhead is approximately 20% for a system containing 10^5 nodes and goes down to 2% for a system containing 10^6 nodes. This result is reasonable because messages get larger and the overhead becomes less significant as system size increases.

4 Two Hop Lookups

The scheme that was presented in the previous section works well for systems as large as a few million nodes. For systems of larger size, the bandwidth requirements of this scheme may become too large for a significant fraction of nodes. In this section, we propose a two-hop scheme. This scheme keeps a fixed fraction of the total routing state on each node and consumes much less bandwidth, and thus scales to a larger system size. We begin by presenting the algorithm design in Section 4.1. Section 4.2 analyzes the bandwidth requirements of this scheme.

4.1 System Design

Our design for a routing algorithm that routes in two hops is based on a structure like that used in the one-hop scheme, with slices, units, and their respective leaders, as described previously.

In addition, every slice leader chooses a group of its own nodes for each other slice; thus there are $k-1$ such groups. Each group is of size l. The groups may be chosen randomly or they may be based on proximity metrics, i.e., each group may be chosen so that its members are dispersed (in terms of network location) in a way that approximates the network spread among all members of the slice.

The slice leader sends routing information about one group to exactly one other slice leader. The information about the group is then disseminated to all members of that slice as in the one hop scheme. Therefore, each node has routing information for exactly l nodes in every other slice. Each node maintains an ordering (e.g., by sending probes) for the l nodes based on network distance to itself. It maintains such an ordering for every slice and thus builds a table of $k-1$ nodes that are close to it, one from every other slice. In addition, every node keeps full routing information about nodes in its own slice.

When a node wants to query the successor of a key, it sends a lookup request to its chosen node in the slice containing the key. The chosen node then examines its own routing table to identify the successor of the key and forwards the request to that node. For the rest of the paper, we shall refer to the chosen intermediate nodes as forwarding nodes.

The information flow in the system is similar to what we saw for the one-hop lookup algorithm in Figure 1. The only difference occurs in what a slice leader sends to other slice leaders in step (**2**). The message it sends to the ith slice leader is empty unless one or more of the l nodes it previously sent to that slice leader have left the system. In that case, it sends information about the nodes that have left the system and the nodes that it chooses to replace them.

When a node learns of different membership for some other slice, it probes the nodes it just heard about and updates its proximity information for that slice.

Tolerating slice and unit leader failure works similarly to the one hop case.

4.2 Analysis

This section presents an analysis of how to parameterize the system to satisfy our goal of fast propagation. As before, our analysis does not take into account overheads of slice and unit leader failure because these events are rare. It also ignores message loss and delay and proximity probes since this simplifies the presentation, and the overhead introduced by probes and by message delays and retransmissions is small compared to other time constants in the system.

As before, our analysis assumes that query targets are located uniformly at random around the ring. It is based on a worst case pattern of queries and notifications. There are two ways in which a query can fail. First, the forwarding node has failed and the querying node is not yet aware of the event. Second, the successor of the key being queried has changed and the forwarding node is not yet aware of the event. The probability of a query failing depends on these events, which may not be independent. Therefore, we assume the upper bound for the failure probability is the sum of the probabilities of these events.

The time taken to spread information about an event within a slice depends on the unit size, and as before, we call it t_{small}. Then the time taken to spread information about an event to all nodes in the system is $t_{tot} = t_{big} + t_{wait} + t_{small}$. Therefore, the average (over locations in the ring) probability of query failure because of leave of forwarding node is approximately $\frac{r}{2 \cdot n} \cdot \left(t_{big} + \frac{t_{wait} + t_{small}}{2}\right)$. The average probability of query failure because of change of key's successor is $\frac{r}{n} \cdot \frac{t_{wait} + t_{small}}{2}$. Therefore, the expected fraction of failed queries is upper bounded by $\frac{r}{n} \cdot (\frac{t_{big}}{2} + \frac{3 \cdot t_{wait}}{2} + \frac{3 \cdot t_{small}}{2})$. Therefore, to ensure that no more than a fraction f of queries fail, we need:

$$2 \cdot t_{big} + 3 \cdot (t_{wait} + t_{small}) \leq \frac{4 \cdot f \cdot n}{r}$$

For example, for a system with 10^8 nodes, with a rate of 20, 000 events/s, and $f = 1\%$, we require that $2 \cdot t_{big} + 3 \cdot (t_{wait} + t_{small}) \leq 200$ seconds. Note that if r is linearly proportional to n, this inequality is independent of n. It is only a function of the desired success rate. We choose $t_{big} = 40$ seconds, $t_{wait} = 1$ seconds and $t_{small} = 30$ seconds. This choice leaves an interval of around 4 seconds for detection of a join or leave event. Given that keep-alive messages are exchanged every second, this implies that the expected size of a unit must be 60. To control upstream bandwidth utilization on slice leaders, we fix the number of units in a slice to 25. This implies that the expected size of a slice should be 1500 and the ring should be divided into $k = n/1500$ slices.

In terms of bandwidth costs, we need to have into account the fact that we are dealing with small messages, so we need to consider protocol overheads. Assume that m bytes will be required to describe an event, and the overhead per message will be v.

There are four types of communication in our system:

1. *Keep-alive messages:* Keep-alives comprise the base level communication between a node and its predecessor and successor. These messages include information about recent events in the node's slice and about exported nodes in other slices. As described in Section 4.1, our system avoids sending redundant information in these messages by controlling the direction of information flow (from unit leader to unit members) and by not sending information across unit boundaries.

   Since keep-alive messages are sent every second, every node that is not on the edge of a unit will send and acknowledge an aggregate message containing, on average, $\frac{r}{k} \cdot (l+1)$ events. The size of this message is therefore $\frac{r}{k} \cdot (l+1) \cdot m + v$ and the size of the acknowledgment is v.
2. *Event notification to slice leaders:* This is identical to the one-hop case. Whenever a node detects an event, it sends a notification to its slice leader. The expected number of events per second in a slice is $\frac{r}{k}$. The downstream bandwidth utilization on slice leaders is therefore $\frac{r \cdot (m+v)}{k}$. Since each message must be acknowledged, the upstream utilization is $\frac{r \cdot v}{k}$.
3. *Messages between slice leaders:* Each message sent from one slice leader to another contains information about changes in exported nodes, if any. The expected message size is, therefore, $\frac{r \cdot l}{n} \cdot t_{big} \cdot m + v$ bytes. During any $t_1 = 40$ seconds period, a slice leader sends this message to all other slice leaders, and receives an acknowledgment from each of them. Since each slice leader receives as much as it gets on average, the upstream and downstream use of bandwidth is symmetric. Therefore, the bandwidth utilization (both upstream and downstream) is

$$\left(\frac{r \cdot l \cdot m}{n} + \frac{2 \cdot v}{40}\right) \cdot (k-1)$$

	Upstream	Downstream
Slice Leader	1.6 Mbps	800 kbps
Unit Leader	1 kbps	530 bps
Ordinary node	530 bps	530 bps

Table 2. Summary of bandwidth use for a system of size 10^8

4. *Messages from slice leaders to unit leaders:* Messages received by a slice leader are batched for one second and then forwarded to unit leaders. In one second, $\frac{r}{k}$ events happen within the slice and $\frac{r \cdot k \cdot l}{n}$ events are exported. There are 25 units per slice, and therefore, the bandwidth utilization is

$$25 \cdot \left(\left(\frac{r}{k} + \frac{r \cdot k \cdot l}{n} \right) \cdot m + v \right)$$

Using these formulas we can compute the load on all nodes for a system of any size and an appropriate choice of l. For a system of size 10^8, we may choose l to be approximately 15. Since slices are large, we expect that this group size will allow each node to be able to find at least one node (in every slice) which is close to it in terms of network proximity even if the groups are populated randomly. This will make the first hop in the lookup a low latency hop, bringing down the total routing delay. If algorithms for clustering nodes on the basis of network proximity are used, then l may be fixed depending on the size and the number of clusters. In this computations we use $m = 10$ bytes and $v = 20$ bytes. Table 2 summarizes the net bandwidth use on each node in a system of size 10^8 having 20,000 events per second, and with $l = 15$. The load on slice leaders increases linearly with the size of the system. Therefore, this scheme would scale up to a system of around half a billion nodes.

5 Evaluation

In this section, we present experimental results obtained with simulations of the one hop and two hop schemes. In the first set of experiments, we used a coarse-grained simulator to understand the overall behavior of the one and two hop systems with tens of thousands of nodes. This simulation scales to approximately 20,000 nodes. In the second set of experiments we use a more fine-grained simulation of the one hop system where the simulation environment could not support more than 2000 nodes.

In both experiments we derived inter-host latencies from Internet measurements among a set of 1024 DNS servers [5]. Note that our experimental results are largely independent of topology since we did not measure lookup latency (this would be influenced by the distance to the forwarding node in the two-hop scheme), and the impact of inter-node latency on query failure rates is minimal, since the latencies are over an order of magnitude smaller than the timeouts used for propagating information.

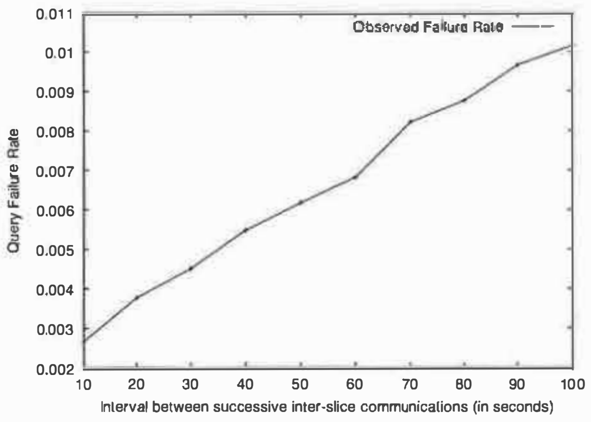


Figure 5. Query Failure Rate in a one hop system of size 20,000 with changing inter-slice communication frequency

5.1 Coarse-grained Simulations

The experiments using the coarse-grained simulator were aimed at validating our analytic results concerning query success rate. The coarse-grained simulator is based on some simplifying assumptions that allow it to scale to larger network sizes. First, it is synchronous: the simulation proceeds in a series of rounds (each representing one second of processing), where all nodes receive messages, perform local processing of the messages, and send messages to other nodes. Second, in this case we did not simulate slice leader failures. Packet losses are also not simulated.

The first set of experiments shows the fraction of successful queries as a function of interslice communication rate. The expected number of nodes in the system is 20,000, the mean join rate is 2 nodes per second, and the mean leave rate is 2 nodes per second. Node lifetime is exponentially distributed. New nodes and queries are distributed uniformly in the ID space. The query rate is 1 query per node per second.

For the one hop scheme, the number of slices is chosen to be 50 and there are 5 units in every slice

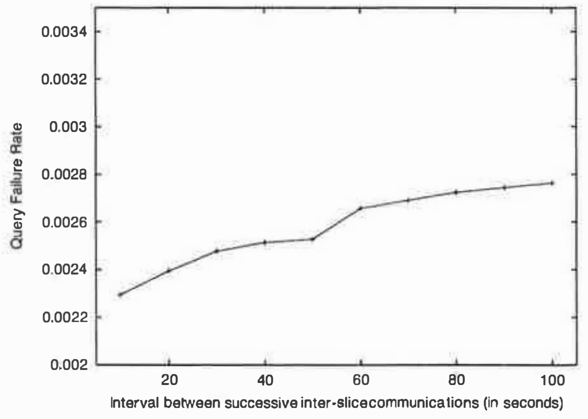


Figure 6. Query Failure Rate in a two hop system of size 20, 000 with changing inter-slice communication frequency

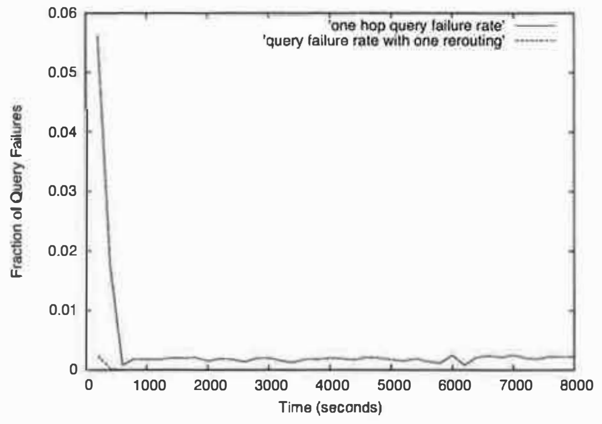


Figure 7. Query Failure Rate in a one hop system of size 2, 000

(these are good choices according to our analysis). The frequency of inter-slice communication varied from 1 every 10 seconds to 1 every 100 seconds.

The results are shown in Figure 5. We can see that the query failure rate grows steadily with the time between inter-slice communication. Note, however, that even with a relatively infrequent communication of once every 100 seconds, we still obtain an average of about a 1% failure rate.

This simulation confirms our expectation that the failed query rate we computed analytically was conservative. We can see that when the inter-slice communication is set to 23 s (the value suggested by our analysis), the query failure rate is about 0.4%, and not 1% as we conservatively predicted in Section 3. The reason why the actual failure rate is lower is because our analysis assumed the worse case where all queries are issued right after membership events occur, and before any events were propagated. In reality, queries are distributed through the time interval that it takes to propagate the information, and by the time some queries are issued, some nodes already have received the most up-to-date information.

Figure 6 shows a similar experiment for the simulation of the two hop scheme. Here the expected number of slices in the system is chosen to be the bandwidth-optimal slice count of 7. Similarly, the number of units per slice is chosen to be 60 (again this choice comes from our analysis). By comparing Figures 5 and 6, we can see that the two hop scheme causes a lower fraction of failed queries than the one hop scheme. This happens for two reasons. In the two hop scheme, the first hop fails only if the forwarding node fails; node joins do not cause routing failures in this case. Also the second hop is more likely to succeed than the single hop in the one hop case, since the choice of target node is made by the forwarding node, which has very up-to-date information about its own slice. These points also explain why the two hop system has much lower sensitivity to change in frequency of inter-slice communications than the one hop system.

5.2 Fine-grained Simulations

In this section we report on simulations of the one hop routing algorithm using p2psim [4], a peer-to-peer protocol simulator where we could implement the complete functionality of the one hop protocol. Using this simulator we are able to explore bandwidth utilization and also the impact of slice and unit leader failures.

In the first experiment we measure the evolution of the query failure rate of a system that grows very fast initially, and then stabilizes to "typical" membership dynamics. We simulate a system with 2000 dynamic nodes with 10 slices and 5 units per slice. The results are shown in Figure 7. In the first 300 seconds of the simulation, all nodes join rapidly. After that the system shows Gnutella-like churn [14] with 24 events per minute. All nodes join by obtaining a routing table from another node; the routing tables then continue to grow as new nodes come in.

After approximately the first 10 minutes, the query failure rate stayed consistently at around 0.2%. We also did experiments to determine the failure rate observed after the query is re-routed *once*. In this case the failure rate settles down to approximately one failure in 10^4 queries, or 0.01%. This

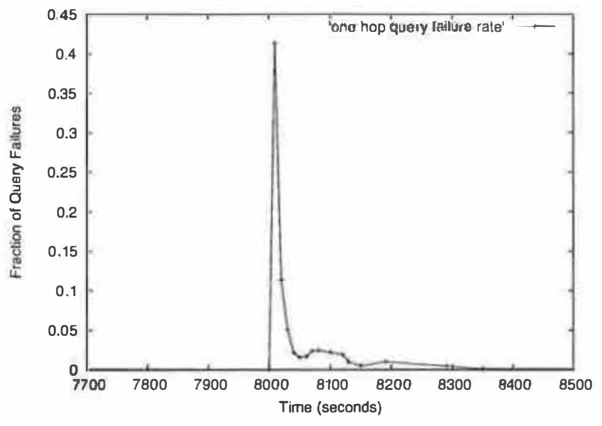


Figure 8. Query Failure Rate in a one hop system of size 2000 after 45% of the nodes crash at t=8000s

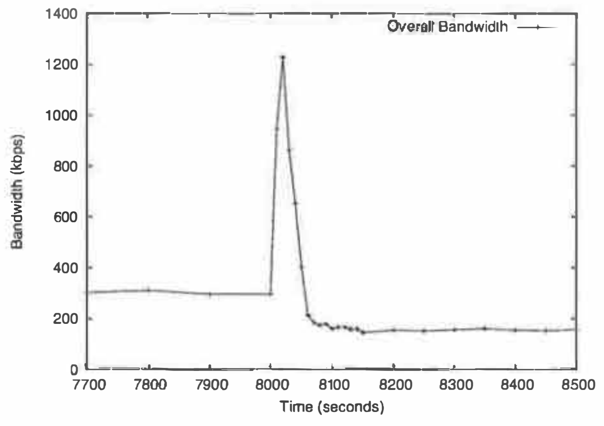


Figure 9. Bandwidth used in a one hop system of size 2000 after 45% of the nodes crash at t=8000s

is because the local information about a slice, especially knowledge of predecessors and successors gets transmitted very rapidly. Thus, 99.99% of the queries are correctly satisfied by two or fewer hops.

Next, we examine the behavior of the system in the presence of a burst of membership departures. Again we simulated a system with 2000 nodes, with 10 slices and 5 units per slice. The query rate was 1 lookup per second per node. At time instant t=8000 seconds, 45% of the nodes in the system were made to crash. These nodes are chosen randomly. Figure 8 shows the fraction of lookups that failed subsequent to the crash. It takes the system about 50 seconds to return to a reasonable query success rate, but it doesn't stabilize at the same query success rate that it had prior to the crash for another 350 seconds. What is happening in this interval is recovery from slice leader failures. The query rate has an important role in slice leader recovery; queries help in restoring stale state by regenerating event notifications that are lost because of slice-leader failures. For example, with a query rate of 1 lookup every 10 seconds, the system did not stabilize below 2% for the length of the simulation (50,000 seconds) while for a query rate of 2 lookups per second, the system stabilized within 300 seconds. This indicates that it may be useful to artificially insert queries in the system during periods of inactivity.

Figure 9 shows the overall bandwidth used in the system in this period. The aggregate bandwidth used in the entire system is around 300 kbps before the crash and settles into around 180 kbps after approximately 150 seconds. (The steady-state bandwidth decreases due to the decrease in the system size.) We can see that while the duration of the spike is similar to that of the spike in query failure rate, bandwidth settles down to expected levels faster than successful lookups. This happens because lookups continue to fail on routing table entries whose notifications are lost in slice leader failures. These failures have to be "re-discovered" by lookups, and then fixed slice-by-slice, which takes longer. While each failed lookup generates notifications and thus increases maintenance bandwidth, at the system size of around 2000 most messages (after the spike settles down) are dominated by the UDP/IP packet overhead. Thus, the overall effect on bandwidth is significantly lower.

We also ran experiments in which we simulated bursts of crashes of different fractions of nodes. We observed that the time periods taken for the lookups and bandwidth to settle down were almost the same in all cases. We expect this to happen because the time taken to stabilize in the system is dependent on the system size, and chosen parameters of unit size and t_{big} which remain the same in all cases.

We also computed the average spike bandwidth. This was measured by computing the average bandwidth used by the entire system in the 50 seconds it took for the system to settle down in all cases. From Figure 10 we see that the bandwidth use grows approximately linearly with the size of the crash. In all cases, the bandwidth consumption is reasonable, given the fact that this bandwidth is split among over a thousand nodes.

6 Discussion

In this section we discuss features that we did not incorporate into our algorithms, but that may be of use in the future.

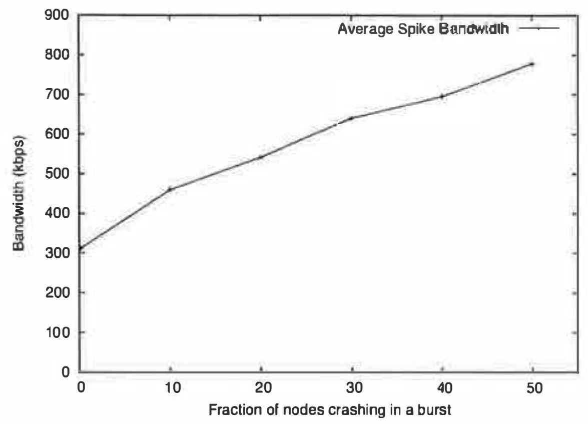


Figure 10. Average spike bandwidth after a fraction of the nodes crash in a burst

6.1 Proximity

Existing peer-to-peer routing algorithms like Pastry [13] and Tapestry [18] carefully exploit inter-node proximity when choosing a node's routing table entries. By trying to populate a node's routing tables with nearby nodes, the routing process is simplified, as shorter routing hops become more likely.

Our one hop scheme does not require proximity for routing, as proximity information is of no use in practice in this scheme. For our two hop scheme, we mentioned in Section 4.1 how proximity can be exploited to improve routing.

However, we can also improve our algorithms by using proximity information when forming our dissemination trees (which in our case are formed by randomly chosen slice and unit leaders). The main improvement comes from improving the dissemination of information within a slice. We think that inter-slice communication is a small part of the overall load, and since the slice leaders are chosen to be well-connected nodes, there is not much point in trying to improve this situation.

For disseminating information within a slice, however, we could improve our current scheme by using an application-level multicast technique that takes proximity into account. Either a peer-to-peer technique (e.g., Scribe [1] or Bayeux [19]) or a traditional technique (e.g., SRM [3] or RMTP [8]) might be appropriate.

6.2 Caching and Load-Balancing

Previous peer-to-peer systems exploited the fact that queries for the same key from different clients have lookup paths that overlap in the final segments, to perform caching of the objects that were returned on the nodes contacted in the lookup path. This provided a natural way to perform load balancing — popular content was cached longer, and thus it was more likely that a client would obtain that content from a cached copy on the lookup path.

Our two hop scheme can use a similar scheme to provide load-balancing and caching. This will lead to popular items being cached at the forwarders, where they can be accessed in one hop; note that an added benefit is the the querying node will usually have good proximity to the forwarder.

Since the one hop scheme doesn't have extra routing steps we can't use them for load balancing. But one-hop routing can be combined with caching schemes to achieve load balancing (and nearby access) if desired. In addition, load balancing might be achieved at the application level by taking advantage of replication. In a peer-to-peer system data must be replicated in order to avoid loss when nodes depart. A query can take advantage of replication to retrieve an item from the replica most proximate to the querying node.

7 Related Work

Rodrigues et al. [12] proposed a single hop distributed hash table but they assumed a much smaller peer dynamics, like that in a corporate environment, and therefore did not have to deal with the difficulties of rapidly handling a large number of membership changes with efficient bandwidth usage. Douceur et al. [2] present a system that routes in a constant number of hops, but that design assumes smaller peer dynamics and searches can be lossy.

Kelips [6] uses $\sqrt{n}$ sized tables per node and a gossip mechanism to propagate event notifications to provide constant time lookups. Their lookups, however, are constant time only when the routing table entries are reasonably accurate. As seen before, these systems are highly dynamic and the accuracy of the tables depends on how long it takes for the system to converge after an event. The expected convergence time for an event in Kelips is $O(\sqrt{n} \times \log^3(n))$. While this will be tens of seconds for small systems of around a 1000 nodes, for systems having 10^5 to 10^6 nodes, it takes over an hour for an event to be propagated through the system. At this rate, a large fraction of the routing entries in each table are likely to be stale, and a correspondingly large fraction of queries would fail on their first attempt.

Mahajan et al. [9] also derive analytic models for the cost of maintaining reliability in the Pastry [13] peer-to-peer routing algorithm in a dynamic setting. This work differs substantially from ours in that the nature of the routing algorithms is quite different – Pastry uses only $O(log\ N)$ state but requires $O(log\ N)$ hops per lookup – and they focus their work on techniques to reduce their (already low) maintenance cost.

Liben-Nowell et al. [7] provide a lower-bound on the cost of maintaining routing information in peer-to-peer networks that try to maintain topological structure. We are designing a system that requires significantly larger bandwidth than in the lower bound because we aim to achieve a much lower lookup latency.

Mizrak et al. [10] present an alternative two-hop routing scheme. In this scheme, all queries are routed through (their equivalent of) slice leaders and ordinary nodes do not exchange state. Our two hop scheme gives the querying node different possibilities for the forwarding node, which allows us to employ clever techniques to decide which forwarding node to use (e.g., based on proximity).

8 Conclusions

This paper questions the necessity of multi-hop lookups in peer-to-peer routing algorithms. We introduce the design of two novel peer-to-peer lookup algorithms. These algorithms route in one and two hops, respectively, unless the lookup fails and other routes need to be attempted. We designed our algorithms to provide a small fraction of lookup failures (e.g., 1%).

We present analytic results that show how we can parameterize the system to obtain reasonable bandwidth consumption, despite the fact that we are dealing with a highly dynamic membership. We present simulation results that support our analysis that the system delivers a large fraction of lookups within one or two hops, depending on the algorithm.

Previous peer-to-peer systems exploited the fact that queries for the same id from different clients have lookup paths that overlap in the final segments, to perform caching of the objects that were returned on the nodes contacted in the lookup path. This provided a natural way to perform load balancing — popular content was cached longer and more often, and it became more likely that a client would obtain that content from a cached copy on the lookup path.

Our two hop algorithm can use a similar scheme to provide load-balancing and caching. We are investigating ways to obtain similar advantages in the one hop scheme.

Currently peer-to-peer systems have high lookup latency and are therefore only well-suited for applications that do not mind high-latency store and retrieve operations (e.g., backups) or that store and retrieve massive amounts of data (e.g., a source tree distribution). Moving to more efficient routing removes this constraint. This way we can enable a much larger class of applications for peer-to-peer systems.

Acknowledgments

We would like to thank our shepherd Ion Stoica, and the anonymous referees for their helpful comments on drafts of this paper.

This research is supported by DARPA under contract F30602-98-1-0237 and by the NSF under Cooperative Agreement ANI-0225660. R. Rodrigues was supported by a fellowship from the Calouste Gulbenkian Foundation, and was previously supported by a Praxis XXI fellowship.

References

[1] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[2] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, July 2002.

[3] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[4] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim: A simulator for peer-to-peer protocols. `http://www.pdos.lcs.mit.edu/p2psim/`.

[5] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[6] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[7] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-First Annual ACM*

Symposium on Principles of Distributed Computing (PODC 2002), 2002.

[8] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *Proceedings IEEE INFOCOM '96*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[9] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[10] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *IEEE Workshop on Internet Applications*, 2003.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, Aug. 2001.

[12] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *Proceedings of the 10th SIGOPS European Workshop*, Sept. 2002.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, Nov. 2001.

[14] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02)*, Jan. 2002.

[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, Aug. 2001.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoeki, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report MIT-LCS-TR-819, MIT, Mar. 2001.

[17] B. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, USA, Mar. 2002.

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

[19] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Network and Operating System Support for Digital Audio and Video, 11th International Workshop, NOSSDAV 2001*, June 2001.

Listen and Whisper: Security Mechanisms for BGP

Lakshminarayanan Subramanian*, Volker Roth++, Ion Stoica*, Scott Shenker*+, Randy H. Katz*

*University of California, Berkeley {lakme,istoica,randy}@cs.berkeley.edu

++ Fraunhofer Institute, Germany vroth@igd.fhg.de

+ ICSI, Berkeley shenker@icsi.berkeley.ed

Abstract

BGP, the current inter-domain routing protocol, assumes that the routing information propagated by authenticated routers is correct. This assumption renders the current infrastructure vulnerable to both accidental misconfigurations and deliberate attacks. To reduce this vulnerability, we present a combination of two mechanisms: *Listen* and *Whisper*. Listen passively probes the data plane and checks whether the underlying routes to different destinations work. Whisper uses cryptographic functions along with routing redundancy to detect bogus route advertisements in the control plane. These mechanisms are easily deployable, and do not rely on either a public key infrastructure or a central authority like ICANN.

The combination of Listen and Whisper eliminates a large number of problems due to router misconfigurations, and restricts (though not eliminates) the damage that deliberate attackers can cause. Moreover, these mechanisms can detect and contain isolated adversaries that propagate even a few invalid route announcements. Colluding adversaries pose a more stringent challenge, and we propose simple changes to the BGP policy mechanism to limit the damage colluding adversaries can cause. We demonstrate the utility of Listen and Whisper through real-world deployment, measurements and empirical analysis. For example, a randomly placed isolated adversary, in the worst case can affect reachability to only 1% of the nodes.

1 Introduction

The Internet is a collection of autonomous systems (AS's), numbering more than 14,000 in a recent count. The inter-domain routing protocol, BGP, knits these autonomous systems together into a coherent whole. Therefore, BGP's resilience against attack is essential for the security of the Internet. BGP currently enables peers to transmit route announcements over authenticated channels, so adversaries cannot impersonate the legitimate sender of a route announcement. This approach, which verifies *who* is speaking but not *what* they say, leaves the current infrastructure extremely vulnerable to both unintentional misconfigurations and deliberate attacks. For example, in 1997 a simple misconfiguration in a customer router caused it to advertise a short path to a large number of network prefixes, and this resulted in a massive black hole that disconnected significant portions of the Internet [14].

To eliminate this vulnerability, several sophisticated BGP security measures have been proposed, most notably S-BGP [25]. However, these approaches typically require an extensive cryptographic key distribution infrastructure and/or a trusted central database (*e.g.,* ICANN [3]). Neither of these two crucial ingredients are currently available, and so these security proposals have not moved forward towards adoption.[1] In this paper we abandon the goal of "perfect security" and instead seek "significantly improved security" through more easily deployable mechanisms. To the end we propose two measures, Listen and Whisper, that require neither a public key distribution nor a trusted centralized database. We first describe the threat model we address and then summarize the extent to which these mechanisms can defend against those threats.

1.1 Threat Model

The primary underlying vulnerability in BGP that we address in this paper is the ability of an AS to create *invalid* routes. There are two types of invalid routes:

Invalid routes in the Control plane: This occurs when an AS propagates an advertisement with a fake AS path (i.e., one that does not exist in the Internet topology), causing other AS's to choose this route over genuine routes. A single malicious adversary can divert traffic to pass through it and then cause havoc by, for example, dropping packets (rendering destinations unreachable), eavesdropping (violating privacy), or impersonating end-hosts within the destination network (like Web servers etc.).

Invalid routes in the Data Plane: This occurs when a router forwards packets in a manner inconsistent with the routing advertisements it has received or propagated; in short, the routing path in the data plane does not match the

[1]There is much debate about whether their failure is due to the lack of a PKI and trusted database, or onerous processing overheads, or other reasons. However, the fact remains that neither of these infrastructures are available, and any design that requires them faces a much higher deployment barrier.

corresponding routing path advertised in the control plane. Mao et al. [27] show that for nearly 8% of Internet paths, the control plane and data plane paths do not match.

Two primary sources of invalid routes are misconfigurations and deliberate attacks. While these are the only sources of invalid routes in the control plane, data plane invalidity can occur additionally due to genuine reasons (e.g. intra/inter-domain routing dynamics [27]). The fact that a sizable fraction of Internet routes are invalid in the data plane motivates the need for separately verifying the correctness of routes in the data plane and not merely focusing on the control plane. Prior works on securing BGP focus primarily on the control plane.

Misconfigurations occur in several forms ranging from buggy configuration scripts to human errors. In the control plane, Mahajan et al. [26] infer that misconfigurations produce invalid route announcements to roughly $200 - 1200$ prefixes every day (roughly $0.2 - 1\%$ of the prefix entries in a typical routing table). Stale routes (not propagating new announcements) and forwarding errors at a router (*e.g.*, lack of forwarding entry) are two other data plane misconfigurations causing invalid routes. While AS's might act in malicious ways on their own, the biggest worry about deliberate attacks comes from adversaries who break into routers. Routers are surprisingly vulnerable; some have *default passwords* [10, 35], others use standard interfaces like telnet and SSH, and so routers share all their known vulnerabilities. For our purposes in this paper, the only difference between a misconfiguration and an attack is that attackers can take active countermeasures (by, for instance, spoofing responses to various probes) while misconfigured routers don't. Deliberate attacks can involve an *isolated adversary* (i.e., a single compromised router) or *colluding adversaries* (i.e., a set of compromised routers). Colluding adversaries have the additional ability to tunnel route advertisements and fake additional links in the topology.

The spectrum of problems we address in this paper can be described, in order of increasing difficulty, as *misconfigurations, isolated adversaries* and *colluding adversaries*. We now describe the extent to which Listen and Whisper provide protection against these threats.

1.2 Level of Protection

Listen detects invalid routes in the data plane by checking whether data sent along routes reaches the intended destination. Whisper checks for consistency in the control plane. While both these techniques can be used in isolation, they are more useful when applied in conjunction. The extent to which they provide protection against the three threat scenarios can be summarized as follows:

Misconfigurations and Isolated Adversaries: Whisper guarantees *path integrity* for route advertisements in the presence of misconfigurations or isolated adversaries; *i.e.,* any invalid route advertisement due to a misconfiguration or isolated adversary with either a fake AS path or with any of the fields of the AS path being tampered (*e.g.*, addition, modification or deletion of AS's) will be detected. Path integrity also implies that an isolated adversary cannot exploit BGP policies to create favorable invalid routes. In addition, Whisper can identify the offending router if it is propagating a significant number of invalid routes. Listen detects reachability problems caused by errors in the data plane, but is only applicable for destination prefixes that observe TCP traffic. However, none of our solutions can prevent malicious nodes already on the path to a particular destination from eavesdropping, impersonating, or dropping packets. In particular, countermeasures (from isolated adversaries already along the path) can defeat Listen's attempts to detect problems on the data path.

Colluding Adversaries: Two colluding nodes can always pretend the existence of a direct link between them by tunneling packets/ advertisements. In the absence of complete knowledge of the Internet topology, these fake links cannot be detected even using heavy-weight security solutions like Secure BGP [24]. While these fake links enable colluding adversaries to propagate invalid routes without being detected, we show that if BGP employs *shortest-path* routing then a large fraction of the paths with fake links can be avoided. On the contrary, colluding adversaries can exploit the current application of BGP policies to mount a large scale attack. To deal with this problem and yet support policy-based routing, we suggest simple modifications to the BGP policy engine which in combination with Whisper can largely restrict the damage that colluding adversaries can cause.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Sections 3 and 4, we describe the whisper and the listen protocols. In Section 5, we present our implementation of Listen and Whisper. In Section 6, we will evaluate several aspects of Listen and Whisper using real-world deployment and security analysis. In Section 7, we discuss the case of colluding adversaries and finally present our conclusions in Section 9.

2 Related Work

In this section, we will present related work as well as try to motivate our work in comparison to previous approaches to this problem. We classify related work based on the threat model.

2.1 Misconfigurations

Traditional approaches to detecting misconfigurations involves correlating route advertisements in the control plane from several vantage points [26, 36]. While these works identify two forms of misconfigurations (origin and export misconfigurations), a fundamental limitation with analyzing BGP streams: *the lack of knowledge of the Internet topology*. Since the topology is not known, these techniques can pinpoint invalid routes only when the destination AS is wrongly specified but not when the path is modified.

Mao *et al.* [27] build an AS-traceroute tool to detect the AS path in the data plane which can be used for data-plane verification. While this tool can detect several forms of invalid routes in the data plane, it is useful for diagnostic purposes only once a problem is detected. Padmanabhan *et al.* [30] propose a secure variant of `traceroute` to test the correctness of a route. However, this mechanism requires a prior distribution of cryptographic keys to the participating AS's to ascertain the integrity and authenticity of traceroute packets. In the context of feedback based routing, Zhu *et al.* [37] proposed a data plane technique based on passive and active probing. The passive probing aspect of this work shares some similarities to our Listen method.

2.2 Dealing with Adversaries

Techniques dealing with adversaries can be classified as *Key distribution based* or *Non-PKI based*.

Key-distribution based: One class of mechanisms builds on cryptographic enhancements of the BGP protocol, for instance the security mechanisms proposed by Smith *et al.* [33], Murphy *et al.* [28], Kent *et al.* [25], and recent work on *Secure Origin BGP* [29]. All these protocols make extensive use of digital signatures and public key certification. More lightweight approaches based on cryptographic hash functions have been proposed *e.g.,* by Hu *et al.* [21, 23] in the context of secure routing in ad hoc networks. However, these mechanisms require prior secure distribution of hash chain elements.

Why not use a PKI-based infrastructure? Public key infrastructures impose a heavy technological and management burden, and have received a fair share of criticism *e.g.,* by Davis [17], Ellison and Schneier [18]. The PKI model has been criticized based on technical grounds, on grounds of a lack of trust and privacy, as well as on principle [17, 18, 16]. Building an Internet wide PKI infrastructure incurs huge costs and has a high risk of failure. Secure-BGP, despite the push by a tier-1 ISP, has been deployed only by a very small number of ISPs after 5 years (though an IETF working group on Secure-BGP exists).

Non-PKI approaches: Non-PKI based solutions offer far less security in the face of deliberate attacks. Some of these mechanisms assume the existence of databases with up to date authoritative route information against which routers verify the route announcements that they receive. The *Internet Routing Registry* [4] and the *Inter-domain Route Validation Service* proposed by Goodell *et al.* [20] belong to this category. Here, the problem is to ascertain the authenticity, completeness, and availability of the information in such a database. First, ISPs only reluctantly submit routing information because this may disclose local policies that the ISPs regard as confidential. Second, the origin authentication of the database contents again demands a public key infrastructure [29]. Third, access to such databases relies on the very infrastructure that it is meant to protect, which is hardly an ideal situation.

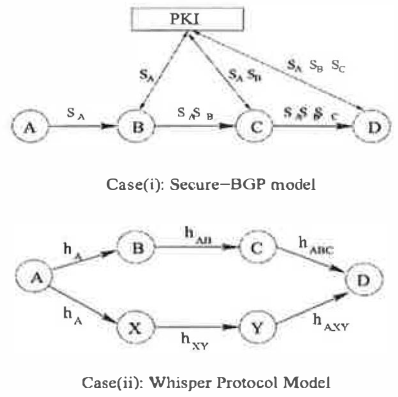


Figure 1: Comparison of the security approach of Whisper protocols with Secure BGP

3 Whisper: Control Plane Verification

In this section, we will describe the whisper protocol, a control plane verification technique that proposes minor modifications to BGP to aid in detecting invalid routes from misconfigured or malicious routers. In this section, we restrict our discussion to the case where an isolated adversary or a single misconfigured router propagates invalid routes. We will discuss colluding adversaries in Section 7.

The Whisper protocol provides the following properties in the presence of isolated adversaries:

1. Any misconfigured or malicious router propagating an invalid route will always a trigger an alarm.
2. A single malicious router advertising more than a few invalid routes will be detected and the effects of these spurious routes will be contained.

3.1 Triggering Alarms vs Identification

The main distinction between our approach and a PKI-based approach is the concept of *triggering alarms* as opposed to *identifying the source of problems*. In Secure-BGP, a router can verify the correctness of a single route advertisement by contacting a PKI and a central authority to test the validity of the signatures embedded in the advertisement . For example, in Figure 1 (Case(i)), each AS X appends an advertisement with a signature S_X generated using its public key. Another AS can use a PKI to check whether S_X is the correct signature of X. In this case, any misconfigured/malicious AS propagating an invalid route will not be able to append the correct signatures of other AS's and can be *identified*.

Without either of these two infra-structural pieces, a router cannot verify a single route advertisement in isolation. The Whisper model is to consider two different route advertisements to the same destination and check whether they are consistent with each other. For example, in Figure 1 Case(ii), each route advertisement is associated with a signature of an AS path. AS D receives two advertisements to destination A and can compare the signatures h_{ABC} and h_{AXY} to check whether the routes (C, B, A) and (Y, X, A) are consistent. When two routes are detected as *inconsistent*, the Whisper protocol can determine that at least one of the routes is invalid. However, it cannot clearly pinpoint the source of the invalid route. Upon detecting inconsistencies, the Whisper protocol can *trigger alarms* notifying operators about the existence of a problem. This method is based on the composition of well-known principles of *weak authentication* as discussed by Arkko and Nikander [11].

Whisper does not require the underlying Internet topology to have multiple disjoint paths to every destination AS. As long as an adversary propagating an invalid route is not on every path to the destination, whisper will have two routes to check for consistency: (a) the genuine route to the destination; (b) invalid path through the adversary.

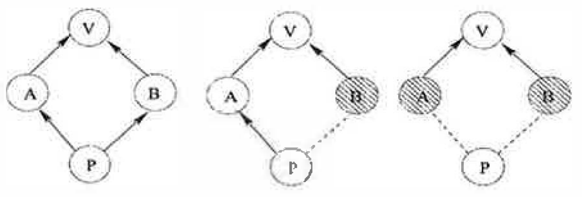
3.2 Route Consistency Testing

A *route consistency test* takes two different route advertisements to the same destination as input and outputs *true* if the routes are consistent and outputs *false* otherwise. Consistency is abstractly defined as follows:

1. If both route announcements are valid then the output is *true*.
2. If one route announcement is valid and the other one is invalid then the output is *false*.
3. If both route announcements are invalid then the output is *true* or *false*.

The key output from a route consistency test is *false*. This output unambiguously signals that *at least one* of the two route announcements is invalid. In this case, our protocols can raise an alarm and flag both the suspicious routes as potential candidates for invalid routes. If the consistency test outputs true, both the routes could either be valid or invalid. Figure 2 depicts the outcomes of a route consistency test for various examples of network configurations.

We will now describe different flavors of route consistency tests of increasing complexity which offer different security guarantees. Conceptually, these constructions introduce a *signature* field in every BGP UPDATE message which is



○ Uncompromised Node ▨ Comrpomised Node ····· Imaginary Path

Figure 2: Different outcomes for a route consistency test. In all these scenarios, the verifying node is V. The verifying node checks whether the two routes it receives to destination P are consistent with each other.

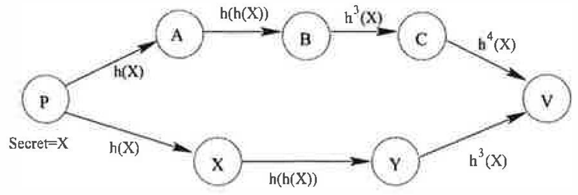


Figure 3: Weak-Split construction using a globally known hash function $h()$

updated by every AS along a path and is used for performing the route consistency test. The origin AS (the originator of a route announcement) of a destination prefix initiates the signature field and every intermediary AS that is not the origin of a destination prefix is required to update the signature field using a cryptographic hash function.

Weak-Split Whisper(WSW): Figure 3 illustrates the weak-split construction using a simple example topology. Weak-Split whisper is motivated by the hash-chain construction used by Hu *et al.* [22, 21] in the context of ad-hoc networks. The key idea is as follows: The origin AS generates a secret x and propagates $h(x)$ to its neighbors where $h()$ is a globally known one-way hash function. Every intermediary AS in the path repeatedly hashes the signature field. An AS that receives two routes r and s of AS hop lengths k and l with signatures y_r and y_s can check for consistency by testing whether $h^{k-l}(y_s) = y_r$.

The security property that the weak-whisper guarantees is: *An independent adversary that is N AS hops away from an origin AS can propagate invalid routes of a minimum length of* $N - 1$ *without being detected as inconsistent.* However, weak split whisper cannot offer path integrity since an adversary can modify the AS numbers along a path without affecting the path length.

The path integrity property requires the whisper protocol to satisfy two properties: (a) a malicious adversary should not be able to reverse engineer the signature field of an AS path; (b) any modification to the AS path or signature field in an advertisement should be detected as an *inconsistency* when tested with a valid route to the same destination.

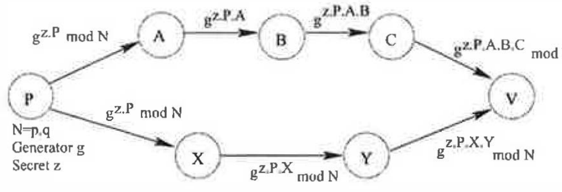


Figure 4: Basic Strong-Split construction using exponentiation under modulo N where $N = p \times q$, a product of two large primes.

3.2.1 RSA-based Strong Split Whisper

Figure 4 shows a hash construction of the whisper signature using the RSA-based strong split whisper(SSW). We use a minor modification of the illustrated example. We will elaborate the three basic operations for this protocol:

generate-signature: The origin AS computes three parameters:. N, g, z. N is chosen as $p \times q$ where p and q are two large primes of the form $2p' + 1$ and $2q' + 1$ where p' and q' are also prime. g is a generator in the prime group Z_p and Z_q and z is a random seed. The signature generated is a tuple $(N, g^z \text{mod} N)$ where only the origin knows the prime factors of N. Similar to RSA, we rely on the fact that an adversary cannot factor N to determine its prime factors.

update-signature: Every AS is associated with a unique AS number which is specified in the path. Assume AS A receives an advertisement with a signature (N, y). A updates this signature to $(N, y^A \text{mod} N)$. In Figure 4, the route announcement for the AS path P, A, B, C, has the signature $(N, g^{z.P.A.B.C} \text{mod} N)$.

verify-signature: We will describe verify-signature using the example in Figure 4. The verifier, V, receives two signatures (N, s_1) and (N, s_2) where $s_1 = g^{z.P.A.B.C} \text{mod} N$ and $s_2 = g^{z.P.X.Y} \text{mod} N$. Given these values and the corresponding AS paths, the verifier outputs the routes to be consistent if:

$$s_1^{X.Y} = s_2^{A.B.C}$$

SSW is similar to the MuHASH construction proposed by Bellare *et al.* [12] for incrementally hashing signatures. The key observation with this construction is: given N and given $g^x \text{mod} N$, an adversary cannot compute $x^{-1} \text{mod} \phi(N)$ (given $N = p \times q$, $\phi(N) = (p-1) \times (q-1)$) and hence cannot remove the signature of previous nodes in the AS path. Additionally, in practice, for every AS B, we use $h(A, B, C)$ in the exponentiation (instead of B) where A and C are the predecessor and successors of B in the route and $h()$ is a one-way hash function. This way, B sets up a binding between itself and its neighbors in the signature. Hence adversaries upstream can neither remove B nor change its position in the AS path.

Common Modulus Problem: One attack on the RSA-based strong split whisper mechanism is the common modulus attack on RSA encryption [32]. If an AS learns two RSA signatures of the form $s^d modN$ and $s^e modN$, an adversary can decipher s provided $gcd(d, e) = 1$. This implies that the RSA-based split whispers will not be able to guarantee *path integrity* in the face of a common modulus attack. This is a fundamental problem with the RSA group and not of the whisper construction. A generalization to the whisper construction requires an Abelian group $(G, \odot)$ which should satisfy two properties: (a) it is computationally infeasible to find the inverse a^{-1} of a given group member a; (b) given two different elements in the group G of the form $s \odot x$ and $s \odot y$, one should not be able to infer s *i.e.*, G does not suffer from the common modulus problem. While the RSA-group satisfies the first property, it does not satisfy the second. Though there exist other Abelian groups like Elliptic curves [13], we are unaware as to whether they suffer from the common modulus problem.

We will now describe two alternate whisper constructions which offer path-integrity and do not have the common modulus problem.

3.2.2 SHA-based Strong Split Whisper

The SHA-based Strong split whisper is an emulation of the previous RSA construction where the exponentiation step is replaced with the SHA one-way hash function [32]. Unlike RSA-based SSW, SHA-based whisper signatures can only be verified by the originator since it does not satisfy the commutativity property of RSA.

Let $h_S()$ represent the SHA one-way hash-function which takes an arbitrary string as input and outputs a 160-bit hash value. A SHA-SSW signature of a route R consists of two parameters: (a) the hash-value of the path; (b) public-key of the originator (as published in the route announcement). In SHA-based SSW, an origin A initiates an announcement to its neighbor B with the signature $h_S(Z, (A, B))$ where Z is a 160-bit nonce and P, its public key. Every intermediary AS B along a path that receives an update from a neighbor A with a SHA signature Y generates the signature $h_S(Y, (A, B, C))$ to its successor C along the path. Hence, a SHA whisper signature is simply a hash signature of the initiator's nonce Z and all neighbor bindings (A, B, C) along a path. The path integrity of SHA signatures follows because the SHA signatures are not: (a) invertible given the one-way hash function property; (b) reproducible by an adversary. Additionally, SHA does not face the common modulus problem.

Consistency Testing: Unlike RSA, only the origin A can verify the correctness of the SHA signature of a path. A node V that receives two routes R, S to origin A performs the following operations for consistency testing. First, if the public keys advertised in routes R and S are inconsistent, then the routes are obviously inconsistent. Second, if the public-keys are the same, V chooses R as its routing path (by fixing its routing table) and sends the encrypted form of S's SHA signature to A querying whether the signature matches the path. A is supposed to send its response to V and signs it with its private key so that V can verify whether A indeed generated the message. Similarly, by setting S as the chosen route, A can verify R's signature. If A responds positively, the routes are deemed consistent. Note that the above test does not make any assumption about the nature of the path from A to V (*i.e.,* symmetric routing is not necessary) since A signs its response using its private key. However, it assumes that at least one valid reverse path exists from A to V. In summary, SHA-based SSW guarantees path-integrity but has the additional complexity of a pair of message exchanges between the verifier and the originator. From an implementation perspective, these messages are routed using normal IP routes and the only modification necessary is an additional signature field in the BGP UPDATE message. We leverage two optimizations to reduce message overhead: (a) The public key of an origin needs to be communicated only once provided future updates use the same consistent public key. (b) Given that the set of distinct routes to a destination AS is relatively stable over time as well as small [15], the SHA signature verification needs to be done only once for each distinct AS path.

3.2.3 Loop Whisper

Loop Whisper is a simple consistency testing strategy which uses AS-level traceroute to check correctness. A verifier V that receives two route advertisements R and S to the same destination A can form an AS-loop involving itself and AS's in R and S. If R and S are completely vertex-disjoint (except the origin A), then the AS-loop is simply $R^{-1}S$ where R^{-1} is the inverse AS-path of R.

Given an AS-loop, the verifier generates a special control message (like an ICMP message) with a nonce and the AS-loop and *source-routes* the message along the loop to test whether the loop exists (nonce is used as a packet identifier). Routing such control messages requires: (a) Each AS should have an additional control mechanism in the routers to handle these specific packets and route them to the neighbor as specified in the source route. (b) Each AS should forward control messages to a neighbor only when a genuine neighbor exists. The second constraint guarantees that if an adversary generates an invalid route with a non-existent path, the loop-test will never succeed. If a loop-test succeeds, two routes are deemed consistent. In summary, while loop whisper guarantees path integrity, it requires at least one router in each AS to support AS-level traceroute.(Note that not all routers need to be modified). From a deployment perspective, SHA-SSW signature based mechanism is easier to deploy than loop whisper.

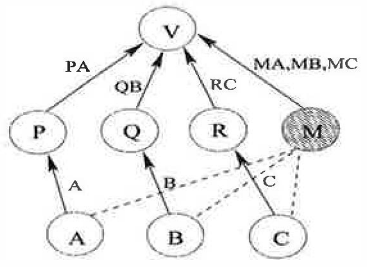


Figure 5: Detecting Suspicious AS's: In this example, M is a malicious AS that propagates 3 invalid routes to 3 different destinations A,B,C. The AS paths in the routes propagated are indicated along the links. The verifier V assigns penalty values of 3,1, 1, 1 to M, A, B, C respectively.

3.3 Containment: Penalty Based Route Selection

Route consistency testing only provides the ability to trigger alarms whenever a node propagates invalid route announcements. We append consistency testing with *penalty based route selection*, a simple containment strategy that attempts to identify suspicious candidates and avoid routes propagated by them. The strategy works as follows: A router counts across destinations how often an AS appears on an invalid route, and assigns this count as a *penalty* value for the AS. The more destinations an adversary affects the higher becomes its penalty and the clearer it stands out from the rest. The route selection strategy is to *choose the route to a destination with the lowest penalty value.*

Consider the topology in Figure 5, where M is a malicious node that propagates 3 invalid route announcements with AS paths MA, MB, MC. By choosing the minimum penalty route, the verifier V can avoid the invalid routes through M since they have a higher penalty value. One key assumption used in this technique is: *The identity of an AS propagating invalid routes is always present in the AS path attribute of the routes.* The identity of every AS is verified by the neighboring AS which receives the advertisement. For example, Zebra's BGP implementation [2] explicitly checks for this constraint for every announcement it receives. BGP should use shared keys across peering links to avoid man in the middle attacks.

Penalties should primarily be viewed as a reasonable first response to detect suspicious candidates and not as a foolproof mechanism. In the presence of an isolated adversary, penalty based filtering can ensure that the effects of the adversary are contained. We believe that penalties is a good mechanism to detect malicious adversaries in customer AS's but should be applied with caution when involving AS's in the Internet core. In particular, penalties are not a good security measure in the presence of colluding adversaries or when the number of independent adversaries is large. For example, multiple adversaries can artificially raise the penalty of an innocent AS by including its AS number in the invalid route.

4 Listen: Data Plane Verification

In this section, we will present the Listen protocol, a data plane verification technique that detects reachability problems in the data plane. Reachability problems can occur due to a variety of reasons ranging from routing problems to misconfigurations to link failures. Listen primarily signals the existence of such problems as opposed to identifying the source or type of a problem.

Data plane verification mechanisms are necessary in two contexts: (a) connectivity problems due to stale routes or forwarding problems are detectable only by data plane solutions like Listen. (b) Blackhole attacks by malicious adversaries already present along a path to a destination. However, proactive malicious nodes can defeat any data plane solution by impersonating the behavior of a genuine end-hosts. The attractive features of Listen are: (a) passive (b) incrementally deployable and standalone solution with no modifications to BGP; (c) quick detection of reachability problems for popular prefixes; (d) low overhead.

The basic form of the protocol described in this section is vulnerable to port scanners generating many incomplete connections. In Section 6.2, we use propose defensive measures against port scanners and motivate them using real world measurements.

4.1 Listening to TCP flows

The general idea of Listen is to monitor TCP flows, and to draw conclusions about the state of a route from this information. The forward and reverse routing paths between two end-hosts can be different. Thus we may observe packets that flow in only one direction. We say that a TCP flow is *complete* if we observe a SYN packet followed by a DATA packet, and we say that it is *incomplete* if we observe only a SYN packet and no DATA packet over a period of 2 minutes (which is longer than the SYN timeout period).

Consider that a router receives a route announcement for a prefix P and wishes to verify whether prefix P is reachable via the advertised route. In the simplest case, a router concludes that the prefix P is reachable if it observes at least one complete TCP flow. On the other hand, the router cannot blindly conclude that a route is unreachable if it does not observe any complete connection. Incomplete connections can arise due to reasons other than just reachability problems. These include: (a) non-live destination hosts; (b) route changes during the connection setup of a single flow i.e. SYN and DATA packets traverse different routes. (c) port scanners generating SYN packets.

Under the assumption that port scanners are not present, detecting reachability problems would be easy. To deal with non-live destinations, a router should notice multiple incomplete connections to N different distinct destination addresses (for a reasonable choice of N). The problem of route changes can be avoided by observing flows over a minimum time period T. Hence, a router can conclude that a prefix is unreachable if during a period t it does not observe a complete TCP flow where t is defined as the *maximum* between: (a) the time taken to observe N or more incomplete TCP flows with different destinations within prefix P; (b) a predefined time period T.

The basic probing mechanism described above suffers from two forms of classification errors: (a) false negatives; (b) false positives. A false negative arises when a router infers a reachable prefix as being unreachable due to incomplete connections. A false positive arises when an unreachable prefix is inferred as being reachable. A malicious end-host can create false positives by generating bogus TCP connections with SYN and DATA packets without receiving ACKs. In Section 6.2, we show how to choose the parameters N and T to reduce the chances of incomplete connections causing false negatives.

4.1.1 Dealing with False Positives

Malicious end-hosts can create false positives by opening bogus TCP connections to keep a router from detecting that a particular route is stale or invalid. Adversaries noticing route advertisements from multiple vantage points (*e.g.*, Routeviews [8]) can potentially notice mis-configurations before routers notice reachability problems. Such adversaries can exploit the situation and open bogus TCP connections.

We propose a combination of *active dropping* and *retransmission checks* as a countermeasure to reduce the probability of false positives.

1. *Active dropping:* Choose a random subset of m_1 packets within a completed connection (or across connections), drop them and raise an alarm if these packets are *not* retransmitted. Alternatively, one can just delay packets at the router instead of dropping them.
2. *Retransmission check:* Sample a different random subset of m_2 packets and raise an alarm if more than 50% of the packets are retransmitted.

An adversary generating a bogus connection cannot decide which packets to retransmit without receiving ACKs. If the adversary blindly retransmits many packets to prevent being detected by Active dropping, the Retransmission check notices a problem. We set a threshold of 50% for retransmission checks assuming that *most* genuine TCP connections will not experience a loss-rate close to 50%.

Consider an adversary that has transmitted k packets in a TCP connection without receiving ACKs to retransmit a fraction, q, of these packets. Let $C(x,y) = \frac{x!}{(x-y)!.y!}$ represent the binomial coefficient for two values x and y. The probability with which the adversary is able to mislead the active dropping test is given by $\frac{C(k\cdot q,m_1)}{C(k,m_1)}$. The probability with which the retransmission check cannot detect an adversary is given by the tail of the binomial distribution $(1-(\sum_{l=m_2/2}^{m_2} C(m_2,l)q^l(1-q)^{m_2-l}))$. Hence the overall probability, p_e, that our algorithm does not detect an adversary is:

$$\frac{C(k\cdot q,m_1)}{C(k,m_1)} \times (1-(\sum_{l=m_2/2}^{m_2} C(m_2,l)q^l(1-q)^{m_2-l}))$$

For a given prefix, the overhead of active dropping can be made very small. By choosing $m_1 = 6$ and dropping only 6 packets across different TCP flows, we can reduce the probability of false positive, p_e, to be less than 0.1%.

This countermeasure is applied only when we notice a discrepancy across different TCP connections to the same destination prefix, *i.e.,* number of incomplete connections and complete connections are roughly the same. In this case, we sample and test whether a few complete connections are indeed bogus.

4.1.2 Detailed Algorithm

Figure 6 presents the pseudo-code for the listen algorithm. The algorithm takes a conservative approach towards determining whether a route is verifiable. Since false positive tests can impact the performance of a few flows, the algorithm uses the constant C_h and C_l to trade off between when to test for false positives. When the test is not applied, we use the fraction of complete connections as the only metric to determine whether the route works. The setting of C_h, C_l depends on the popularity of the prefixes. Firstly, we apply the false positive tests only for popular prefixes *i.e.,* $C_l = 0$ for non-popular prefixes. For a popular prefix, we choose a conservative estimate of C_h (closer to 1) *i.e.,* a large fraction of the connections have to complete in order to conclude that the route is verifiable. On the other hand, if we observe that a reasonable fraction of combination of incomplete connections, we apply the false positive test to 2 sampled complete connections. The user has choice in tuning C_l based on the total number of false positive tests that need to be performed. For non-popular prefixes, the statistical sample of connections is small. For such prefixes, we set the value of C_h to be small.

```
procedure LISTEN(P,T,N)
Require: Prefix $P$, time period $T$, number of unique
    destinations $N$
 1: $t_0$ = time at which first SYN packet observed
 2: wait until |flows with distinct dest. in $P| \geq N$
 3: wait till clock time $> t_0 + T$
 4: {Clean the data-set}
 5: For every pair of IP addresses $(src, dst)$ observed
 6: if at least a single connection has completed then
 7:     Add sample $(src, dst, complete)$
 8: else
 9:     Add sample $(src, dst, incomplete)$
10: end if
11: {Constants $C_h, C_l$ must be determined in prac-
    tice}
12: if fraction of complete connections $> C_h$ then
13:     return "route is verifiable"
14: end if
15: if at least one connection completes then
16:     if fraction of complete connections $< C_l$ then
17:         {Test for false positive}
18:         sample 2 future complete TCP flows towards
            $P$
19:         apply active dropping and retransmission
            checks
20:         if test is successful then
21:             return "route is verifiable"
22:         else
23:             return "route is not verifiable"
24:         end if
25:     end if
26: end if
```

Figure 6: Pseudo-code for the probing algorithm.

5 Implementation

In this section, we will describe the implementation of Listen and Whisper and their overhead characteristics.

5.1 Whisper Implementation

In this section, we will only focus on the implementation of the strong split whisper protocol (RSA variant). The SHA variant requires a modification to the hash function we use in our code. [2] The whisper implementation contains two basic components: (a) a stand alone whisper library which performs the cryptographic operations used in the protocol. (b) a Whisper-BGP interface which integrates the whisper functions into a BGP implementation. We implemented the Whisper library on top of the *crypto* library supported by OpenSSL development version 0.9.6b-33. We integrated this library with the Zebra BGP router implementation version 0.93b [2]. Our Whisper implementation works on Linux and FreeBSD platforms.

5.1.1 Whisper Library

The structure of a basic Whisper signature is:

```
typedef struct {
  BIGNUM *seed;
  BIGNUM *N;
}Signature;
```

[2]The additional control messages in SHA-based SSW are data-plane messages and are not incorporated in the code.

BIGNUM is a basic data structure used within the OpenSSL crypto library to represent large numbers. The whisper library supports these three functions using the Signature data structure:

```
1: generate_signature(Signature *sg);
2: update_signature(Signature *sg, int asnumber, int
   position);
3: verify_signatures(Signature *r, Signature *s,int
   *aspath_r, int *aspath_s);
```

These functions exactly map to the three whisper operations described earlier in Section 3.2.1. The main advantage of separating the whisper library from the whisper-BGP interface is modularity. The whisper library can be used in isolation with any other BGP implementation sufficiently different from the Zebra version.

5.1.2 *Integration with BGP*

The Whisper protocol can be integrated with BGP without changing the basic packet format of BGP. BGP uses $32-bit$ community attributes which are options within UPDATE messages that can be leveraged for embedding the signature attributes. This design offers us many advantages over updating a version of BGP. First, a single update message can have several community attributes and one can split a signature among multiple community attributes. Second, a community attribute can be set using the BGP configuration script to allow operators the flexibility to insert their own community attribute values. In a similar vein, one can imagine a stand-alone whisper library computing the signatures and a simple interface to insert these signatures within the community attributes. Third, one can reserve a portion of the community attribute space for whisper signatures. In today's BGP, community values can be set to any value as long as they are interpreted correctly by other routers. An RSA-SSW uses 2048 bits per signature field, while SHA-SSW needs $1184 = 160 + 1024$ bits for the SHA signature and public key.

5.2 Listen Implementation

We implemented the passive probing component of *Listen* (i.e. without active dropping) in about 2000 lines of code in C and have ported the code to Linux and FreeBSD operating systems. The current prototype uses the *libpcap* utility [5] to capture all the packets off the network. This form of implementation has two advantages: (a) is stand-alone and can be implemented on any machine (need not be a router) which can sniff network traffic; (b) does not require any support from router vendors. Additionally, one can execute *bgpd* (Zebra's BGP daemon [2]) to receive live BGP updates from a network router. For faster line-rates (e.g. links in ISPs), *listen* should be integrated with hardware or packet probing software like Cisco's Netflow [1]. The current implementation cannot support false positive tests since the code can only passively observe the traffic but cannot actively drop packets (since this does not perform the routing functionality).

Operation	512-bit	1024-bit	2048-bit
update_signature	0.18 msec	0.45 msec	1.42 msec
verify_signatures	0.25 msec	0.6 msec	1.94 msec
generate_signature	0.4 sec	8.0 sec	68 sec

Table 1: Processing overhead of the Whisper operations on a 1.5 Ghz Pentium IV with 512 MB RAM.

In our implementation, the complexity of listening to a TCP flow is of the same order as a route lookup operation. Additionally, the state requirement is $O(1)$ for every prefix. We maintain a small hash table for every prefix entry corresponding to the (src,dst) IP addresses of a TCP flow and a time stamp. While a SYN packet sets a bit in the hash table, the DATA packet clears the bit and record a complete connection for the prefix. Using a small hash table, we can crudely estimate the number of complete and incomplete connections within a time-period T. Additionally, we sample flows to reduce the possibility of hash conflicts. This implementation uses simple statistical counter estimation techniques used to efficiently maintain statistics in routers. Hence, the basic form of Listen can be efficiently implemented in the fast path of today's routers.

Deployment: We deployed our *Listen* prototype to sniff on TCP traffic to and from a $/24$ prefix within our university. Additionally, we received BGP updates from the university campus router and constructed the list of prefixes in the routing table used by the edge router. The tool only needs to know the list of prefixes in the routing table and assumes a virtual route for every prefix. The Listen tool can report the list of verifiable and non-verifiable prefixes in real time. Additionally, the *Listen* algorithm is applied only by observing traffic in one direction (either outbound or inbound).

5.3 Overhead Characteristics

Overhead of Whisper: One of the important requirements of any cryptography based solution is low complexity. We performed benchmarks to determine the processing overhead of the Whisper operations. Table 1 summarizes the average time required to perform the whisper operations for 3 different key sizes: $512-$ bit, $1024-$bit and $2048-$bit. As the key size increases, the RSA-based operations offer better security. Security experts recommend a minimum size of 1024 bit keys for better long-term security.

We make two observations about the overhead characteristics. First, the processing overhead for all these key sizes are well within the limits of the maximum load observed at routers. For 2048 bit keys, a node can process more than 42, 000 route advertisements within 1 minute. In comparison, the maximum number of route advertisements observed at a Sprint router is 9300 updates every minute [9]. For 1024 bit keys, Whisper can update and verify over 100, 000 route advertisements per minute. Second, *generate_signature()* is an expensive operation and can consume more than 1 sec per operation. However, this operation is performed only once over many days.

Overhead of Listen: By analyzing route updates for over 17 days in Routeviews [8], we observed that 99% of the routes in a routing table are stable for at least 1 hour. Based on data from a tier-1 ISP, we find that a router typically observes a maximum of 20000 active prefixes over a period of 1 hour *i.e.,* only 20000 prefixes observe any traffic. If the probing mechanism uses a statistical sample of 10 flows per prefix, the overhead of probing at the router is negligible. Essentially, the router needs to process 200000 flows in 3600 sec which translates to monitoring under 60 flows every second (equivalent to $O(60)$ routing lookups). Even if the number of active prefixes scales by a factor of 10, current router implementations can easily implement the passive probing aspect of Listen.

Active dropping and retransmission checks are applied only in the IP slow path and are invoked only when a prefix observes a combination of both incomplete and complete connections. To minimize the additional overhead of these operations, we restrict these checks to a few prefixes.

6 Evaluation

In this section, we evaluate the key properties of Listen and Whisper. Our evaluation is targeted at answering specific questions about Listen and Whisper:

1. How much security can Whisper provide in the face of isolated adversaries?
2. How useful is Listen in the real world? In particular, can it detect reachability problems?
3. How does Listen react in the presence of port scanners? How does one adapt to such port scanners?

We answer question (1) in Section 6.1, questions (2),(3) in Section 6.2. Our evaluation methodology is two-fold: (a) empirically evaluate the security properties of Whisper; (b) use a real-world deployment to determine usefulness of Listen. To evaluate the security properties of Whisper, it is necessary to determine the effects of the worst-case scenario which is better quantified using an empirical evaluation.

We collected the Internet AS topology data based on BGP advertisements observed from 15 different vantage points over 17 days including Routeviews [8] and RIPE [7]. The policy-based routing path between a pair of AS's is determined using customer–provider and peer–peer relationships, which have been inferred based on the technique used in [34].

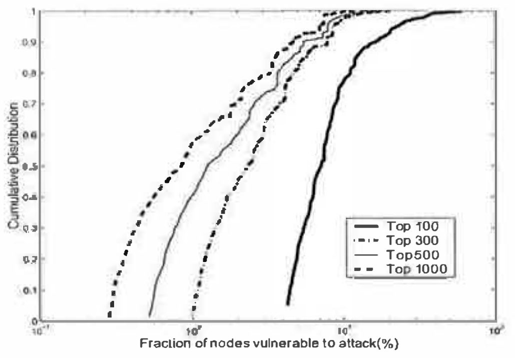


Figure 7: Effects of penalty based route selection

6.1 Whisper: Security Properties against Isolated Adversaries

In this section, we quantify the maximum damage an isolated adversary can inflict on the Internet given that Strong Split Whisper is deployed. Since SHA-based SSW offers path integrity, an isolated adversary cannot propagate invalid routes without raising alarms unless there exists no alternate route from the origin to the verifier (i.e. adversary is present in all paths from the origin to the Internet).

Given an adversary that is willing to raise alarms, we analyzed how many AS's can one such adversary affect. In this analysis, we exclude cases where the adversary is already present in the only routing path to a destination AS. We use penalty based route selection as the main defense to contain the effects of such invalid routes. We assume that in the worst-case, an adversary compromising a single router in an AS is equivalent to compromising the entire AS especially if all routers within the AS choose the invalid route propagated by the compromised router.

Let M represent an isolated adversary propagating an invalid route claiming direct connectivity to an origin AS O. AS V is said to be *affected* by the invalid route if V chooses the route through M rather than a genuine route to O either due to BGP policies or shorter hop length. Based on common practices, we associate all AS's with a simple policy where customer routes have the highest preference followed by peers and providers [19]. Given all these relationships, we define the *vulnerability* of an origin AS, O, as $V(O, M)$ to be the maximum fraction of AS's, M can affect. Given an isolated adversary M, we can quantify the worst-case effect that M can have on the Internet using the *cumulative distribution* of $V(O, M)$ across all origin AS's in the Internet.

	Number of Reachability Problems	Probability of False Negatives
Outbound	235	0.93%
Inbound	343	0.37%

Table 2: Listen: Summary of Results

With AS's deploying penalty based route selection as a defense, we expect the vulnerability $V(O, M)$ to reduce. We study how the cumulative distribution of $V(O, M)$ for a single adversary M varies as a function of how many AS's deploy penalty based route selection. We consider the scenario where the top n ISPs deploy penalty based route selection (based on AS degree). Figure 7 shows this cumulative distribution for for different values of $n = 100, 300, 500$ and 1000. These distributions are averaged across all possible choices for M.

We make the following observations. First, a median value of 1% for $n = 1000$ indicates that a randomly located adversary can affect at most 1% of destination AS's by propagating bogus advertisements assuming that the top 1000 ISPs use penalties. This is orders of magnitude better that what the current Internet can offer where a randomly located adversary can on an average affect nearly 30% of the routes (repeat the same analysis without SSW) to a randomly chosen destination AS.

Second, in the worst case, a single AS can at most affect 8% of the destination AS's for $n = 1000$. 8% is a limit imposed by the structure of the Internet topology since it represents the size of the largest connected without the top 1000 ISPs. One malicious AS in this component can potentially affect other AS's within the same component.

Third, if all provider AS's use penalties for route selection, the worst case behavior can be brought to a much smaller value than 8%. Additionally, there is very little benefit in deploying penalty based route selection in the end-host networks since they are not transit networks and typically are sources and sinks of route advertisements. Hence, any filtering at these end-hosts only protects themselves but not other AS's.

To summarize, the Whisper protocol in conjunction with penalty based route selection can guarantee that a randomly placed isolated adversary propagating invalid routes can affect at most 1% of the AS's in the Internet topology.

6.2 Listen: Experimental Evaluation

In this section, we describe our real-world experiences using the Listen protocol. We make two important observations from our analysis. First, we found that a large fraction of incomplete TCP connections are *spurious i.e.,* not indicative of a reachability problem. We show that by adaptively setting the parameters T, N of our listen algorithm

Number of end-hosts behind /24 network	28
Number of days	40
Total No. of TCP connections	994234
No. of complete connections	894897
No. of incomplete connections	99337
Average Routing Table Size	123482
Total No. of Active Prefixes	11141
Average No. of Active Prefixes per hour	141
Average No. of Active Prefixes per day	2500-3000
Verifiable Prefixes	9711
Prefixes with perennial problems	42

Table 3: Aggregate characteristics of Listen from the deployment

we can drastically reduce the probability of such false negatives due to such connections. Second, we detect several reachability problems using Listen including specific misconfiguration related problems like forwarding errors. Table 2 presents a concise summary of the results obtained from our deployment. We detected reachability problems to 578 different prefixes with a very false negative probabilities of 0.95% and 0.37% respectively due to spurious outbound and inbound connections.

We will now describe our deployment experience in greater detail. In our testbed, we use three active probing tests to verify the correctness of results obtained using Listen: (a) ping the destination; (b) traceroute and check whether any IP address along in the path is in the same prefix as the destination; (c) perform a port 80 scan on the destination IP address. These tests are activated for every incomplete connection. We classify an incomplete connection as having a reachability problem only if all the three probing tests fail. We classify an incomplete connection as a *spurious connection* if one of the probing techniques is able to detect that the route to a destination prefix works. A spurious TCP connection is an incomplete connection that is not indicative of a reachability problem.

Table 3 presents the aggregate characteristics of the traffic we observed from a /24 network for over 40 days. In reality, we found that nearly 10% of the connections are incomplete of which a large fraction of these connections are spurious (91% inbound and 63% outbound). A more careful observation at the spurious connections showed that nearly 90% of spurious inbound connections are due to port scanners and worms. The most prominent ones being the Microsoft NetBIOS worm and the SQL server worms [6]. Spurious outbound connections occur primarily due to failed connection attempts to non-live hosts and attempts to access a disabled ports of other end-hosts (*e.g.,* telnet port being disabled in a destination end-host).Given this alarmingly high number of spurious connections, we propose defensive measures to reduce the probability of false negatives due to such connections.

6.2.1 *Defensive Measures to reduce False Negatives*

In this section, we show that one can adaptively set the parameters N, T in the listen algorithm to drastically reduce the probability of false negatives due to spurious TCP connections. In particular, we show that by adaptively tuning the minimum time period, T, one can reduce false negatives due to port scanners and by tuning the number of distinct destinations, N, one can deal with non-live hosts.

Given the nature of incomplete connections in our testbed, we use outbound incomplete connections as a test sample for non-live hosts and inbound connections as the test sample for port scanners and worms. In both inbound and outbound, we restricted our samples to only those connections which are known to be false negatives.

Setting T: One possibility is to choose an interval T large enough such that the router will notice at least one genuine TCP flow during the interval. Such a value of T will depend on the popularity of a prefix. The popularity of a prefix, $pop(P)$, is defined as the mean time between two complete TCP connections to prefix P. We can model the arrival of TCP connections as a Poisson process with a mean arrival rate as $1/pop(P)$ [31]. Given this, we can set the value of $T = 4.6 \times pop(P)$ to be 99% certain that one would experience at least one genuine connection within the period T. To have a 99.9% certainty, one needs to set $T = 6.9 \times pop(P)$. For prefixes that hardly observe any traffic, the value of T will be very high implying that port scanners generating incomplete connections to such prefixes will not generate any false alarms.

From our testbed, we determine the mean separation time between the arrival of two incoming connections to be $pop(P) = 34.1$ sec. By merely setting $T = 156.8$ to achieve 99% certainty, we could reduce the probability of false negatives in Listen from 91.83% to 0.37%. Throughout the entire period of measurement, only during 8 periods of 156 seconds each did we verify incorrectly that the local prefix is not reachable.

Setting N: The choice of an appropriate value of N trades off between minimizing the false negative ratio due to non-live hosts and the number of reachability problems detected. In our testbed, we noticed that by merely setting $N = 2$, we can significantly reduce the false negative ratio in outbound connections from 63% to less than 1%. However, Listen reported only 35 out of 663 potential prefixes to have routing problems. For several /24 prefixes, we observed TCP connections to only a single host and by setting $N = 2$, we tend to omit these cases. In practice, the value of N is dependent on the diversity of traffic to a destination prefix and the traffic concentration at a router. For many /24 prefixes, we need to set $N = 1$. For /8 and /16 prefixes, one can choose larger values of $N = 4$ or $N = 5$ provided the prefix observes diversity in the traffic.

Type of problem	Number of Prefixes
Routing Loops	51
Forwarding Errors	64
Generic (forward path)	146
Generic (reverse path)	317

Table 4: The number of prefixes affected by different types of reachability problems.

6.2.2 *Detected Reachability Problems*

Among the reachability problems detected by Listen, two specific types of routing problems (as detected by active probing) include: *routing loops* and *forwarding errors* due to unknown IP addresses. We detected routing loops using traceroute and inferred forwarding errors using the routing table entries at the University exit router. A forwarding error arises when the destination IP address in a packet is a genuine one but the router has no next hop forwarding entry for the IP address. This can potentially arise due to staleness of routes. Table 4 summarizes the number of prefixes affected by each type of problem. In particular, we observe routing loops to 51 different prefixes and forwarding errors to 64 different prefixes. Additionally, Listen detected 463 prefixes having other forms of reachability problems.

To cite a few examples of reachability problems we observed: (a) A BGP daemon within our network attempted to connect to another such daemon within the destination prefix 193.148.15.0/24. The route to this prefix was perennially unreachable due to a routing loop. (b) The route to Yahoo-NET prefix 207.126.224.0/20 was fluctuating. During many periods, the route was detected as unavailable.

7 Colluding Adversaries

Additional to acting as a group of isolated adversaries, colluding adversaries can tunnel advertisements and secrets between them and create invalid routes with fake AS links without being detected by the Whisper protocols. These invalid routes are not detectable even with a PKI unless the complete topology is known and enforced. Despite the limitation, we can provide protective measures for avoiding these invalid routes.

Given the hierarchical nature and the skewed structure of the Internet topology, the invalid paths from colluding adversaries not detectable by the Whisper tend to be longer in AS path length. This is because, a normal route would traverse the Internet core (tier-1 + tier-2 ISPs) once while a consistent invalid route through 2 colluding adversaries traverses the Internet core twice (since the adversary cannot remove any AS from the path). Hence, by choosing the shortest path we have a better chance of avoiding the invalid route. Figures 8, 9 and 10, illustrates this effect of colluding adversaries for 3 scenarios: (a) the current Internet with

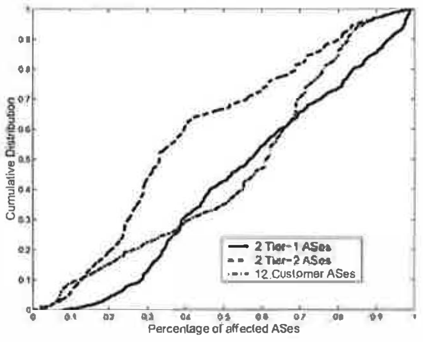


Figure 8: The effects of colluding adversaries in the current Internet.

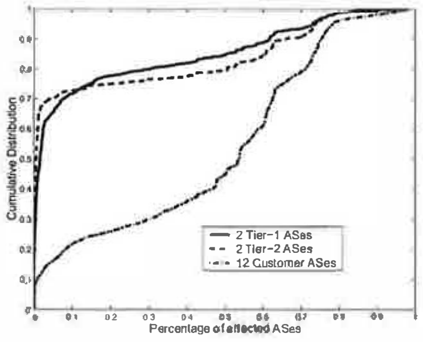


Figure 9: Effects of colluding adversaries with whisper + policy routing.

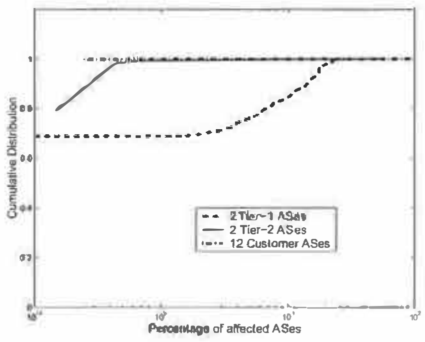


Figure 10: Effect of colluding adversaries with whisper + shortest path routing

no protection; (b) whisper protocols with policy routing; (c) whisper protocols with shortest path routing. All these graphs show the cumulative distribution of the vulnerability metric (defined in Section 6.1) for a set of colluding malicious adversaries. We specifically consider three cases: (a) 2 colluding tier-1 AS's; (b) 2 colluding tier-2 AS's (c) 12 colluding customer AS's.

We make two observations. First, 12 randomly compromised customer routers can inflict the same magnitude of damage as that of two tier-1 nodes illustrating the effect of colluding adversaries in the current Internet. Typically, customer AS's are easier to compromise since many of them are unmanaged. Second, whisper protocols with shortest path routing drastically reduces the possibility of colluding adversaries (in comparison to policy routing) propagating invalid routes without triggering alarms. In particular, even when 12 customer AS's are compromised, the effect on the Internet routing is negligible.

Whisper protocols with policy routing offers much lesser protection since BGP tends to choose routes based on the *local preference*. The typical policy convention based on stable routing and economic constraints is to prefer customer routes over peer and provider routes [19]. This preference rule increases the vulnerability of BGP to pick consistent invalid routes from customers over potentially shorter routes through peers /providers. In principle, this problem also exists in S-BGP. To strike a middle ground between the flexibility of policy routing and this vulnerability, we propose a simple modification to the policy engine: *Do not associate any local preference to customer routes that have an AS path length greater than* 2 (any route from a pair of colluding route should have a minimum path length of 3). We believe that this modification to BGP policies should have little impact on current operation since most customer routes today have a path length less than 3.

To summarize, whisper protocols in combination with the modified policies (emulating shortest path routing) can largely restrict the damage of colluding adversaries.

8 Discussion

We now discuss three specific issues not covered earlier.

Hijacking unallocated prefixes: With the deployment of Whisper, a malicious adversary can still claim ownership over unallocated address spaces without triggering alarms by propagating bogus announcements. One way of dealing with this problem is to request ICANN [3] to specifically advertise unallocated address spaces with its own corresponding Whisper signatures whenever it notices an advertisement for an unallocated prefix. Additionally, to avoid a DoS attack on ICANN for such prefixes, routers should not maintain forwarding entries for these prefixes.

Route Aggregation: Whenever an AS aggregates several route advertisements into one, it is required to perform one of the following operations to maintain the consistency of the aggregated route: (a) Append the individual signatures corresponding to each advertisement so that an upstream AS can match at least one of the signatures with the whisper signatures for alternate routes to sub-prefixes. (b) If the AS owns the entire aggregated prefix (common form of aggregation in BGP), ignore the whisper signatures in the sub-prefixes and append its own whisper signature.

Other types of security attacks: Other than propagation of invalid routes, one can imagine other forms of routing attacks or misconfiguration errors which may result in routing loops, persistent route oscillations or convergence problems. Such problems are out of the scope of this paper.

9 Conclusions

In this paper we consider the problem of reducing the vulnerability of BGP in the face of misconfigurations and malicious attacks. To address this problem we propose two techniques: Listen and Whisper. Used together these techniques can detect and contain invalid routes propagated by isolated adversaries, and a large number of problems due to misconfigurations. To demonstrate the utility of Listen and Whisper, we use a combination of real world deployment and empirical analysis. In particular, we show that

Listen can detect unreachable prefixes with a low probability of false negatives, and that Whisper can limit the percentage of nodes affected by a randomly placed isolated adversary to less than 1%. Finally, we show that both Listen and Whisper are easy to implement and deploy. Listen is incrementally deployable and does not require any changes to BGP, while Whisper can be integrated with BGP without changing the packet format.

Acknowledgments

The anonymous reviewers and Amin Vahdat, our shepherd provided us with invaluable feedback which helped substantially towards improving the quality of the paper. Tom Anderson, Alexandros Batsakis, Anand Desai, Nick Feamster, Mark Handley, Chris Karlof, Ratul Mahajan, Satomi Okazaki, Vern Paxson, Adrian Perrig, Jennifer Rexford, Dawn Song, Doug Tygar and David Wagner provided several technical comments on this work. Krishna Gummadi and Konstantina Papagianakki provided us with valuable data for empirically evaluating our Listen algorithm. Several students in Berkeley and Johns Hopkins University read earlier drafts of this paper and provided very useful feedback. The authors would like to thank them all.

References

[1] Cisco ios netflow. http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml.
[2] Gnu zebra router implementation. http://www.zebra.org/.
[3] Internet Corporation for Assigned Names and Numbers. http://www.icann.org/.
[4] Internet routing registry. http://www.irr.net/. Version current January 2003.
[5] libpcap utility. http://sourceforge.net/projects/libpcap.
[6] Microsoft port 1433 vulnerability. http:/lists.insecure.org/lists/vuln-dev/2002/Aug/0073.html.
[7] Ripe ncc. http://www.ripe.net.
[8] Routeviews. http://www.routeviews.org/.
[9] Sprint IPMON project. http://ipmon.sprint.com/.
[10] Trends in dos attack technology. http://www.cert.org/archive/pdf/DoS_trends.pdf.
[11] J. Arkko and P. Nikander. How to authenticate unknown principals without trusted parties. In *Proc. Security Protocols Workshop 2002*, April 2002.
[12] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. volume 1223 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
[13] I. Blake, G. Serossi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 2000.
[14] V. J. Bono. 7007 explanation and apology. http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html.
[15] D. F. Chang, R. Govindan, and J. Heidemann. Temporal and topological characteristics of BGP path changes. In *IEEE ICNP*, 2003.
[16] R. Clarke. Conventional public key infrastructure: An artefact ill-fitted to the needs of the information society. Technical report. http://www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html.
[17] D. Davis. Compliance defects in public key cryptography. In *Proc. 6th USENIX Security Symposium*, 1996.
[18] C. Ellison and B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000. Available online at URL http://www.counterpane.com/pki-risks.html.
[19] L. Gao and J. Rexford. Stable internet routing without global coordination. In *IEEE/ACM Transactions on Networking*, 2001.
[20] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *Proc. of NDSS*, San Diego, CA, USA, Feb. 2003.
[21] Y. Hu, D. B. Johnson, and A. Perrig. SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks. In *Proc. of WMCSA*, June 2002.
[22] Y. Hu, A. Perrig, and D. B. Johnson. Wormhole detection in wireless ad hoc networks. Technical Report TR01-384, Department of Computer Science, Rice University, Dec. 2001.
[23] Y. Hu, A. Perrig, and D. B. Johnson. Efficient security mechanisms for routing protocols. In *Proc. of NDSS'03*, February 2003.
[24] S. Kent, C. Lynn, and K. Seo. Design and analysis of the Secure Border Gateway Protocol (S-BGP). In *Proc. of DISCEX '00*.
[25] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol (Secure-BGP). *IEEE Journal on Selected Areas of Communications*, 18(4):582–592, Apr. 2000.
[26] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfigurations. In *Proc. ACM SIGCOMM Conference*, Pittsburg, Aug. 2002.
[27] Z. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an accurate AS-level traceroure tool. In *ACM SIGCOMM*, 2003.
[28] S. Murphy, O. Gudmundsson, R. Mundy, and B. Wellington. Retrofitting security into Internet infrastructure protocols. In *Proc. of DISCEX '00*, volume 1, pages 3–17, 1999.
[29] J. Ng. Extensions to BGP to support Secure Origin BGP (sobgp). Internet Draft draft-ng-sobgp-bgp-extensions-00, Oct. 2002.
[30] V. N. Padmanabhan and D. R. Simon. Secure traceroute to detect faulty or malicious routing. In *Proc. HotNets-I*, 2002.
[31] V. Paxson and S.Floyd. Wide area traffic: Failure of poisson modeling. In *Proc. ACM SIGCOMM*, 1994.
[32] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 1996.
[33] B. Smith and J. Garcia-Luna-Aceves. Securing the Border Gateway Routing Protocol. In *Proc. Global Internet '96*, London, UK, November 1996.
[34] L. Subramanian, S.Agarwal, J.Rexford, and R. H. Katz. Characterizing the Internet hierarchy from multiple vantage points. In *IEEE INFOCOM*, New York, 2002.
[35] R. Thomas. http://www.cmyru.com.
[36] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. An analysis of BGP multiple origin AS (MOAS) conflicts. In *ACM SIGCOMM IMW*, 2001.
[37] D. Zhu, M. Gritter, and D. Cheriton. Feedback based routing. In *Proc. of HotNets-I*, October 2002.

Measurement and Analysis of Spyware in a University Environment

Stefan Saroiu, Steven D. Gribble, and Henry M. Levy
Department of Computer Science & Engineering
University of Washington
{tzoompy,gribble,levy}@cs.washington.edu

Abstract

Over the past few years, a relatively new computing phenomenon has gained momentum: the spread of "spyware." Though most people are aware of spyware, the research community has spent little effort to understand its nature, how widespread it is, and the risks it presents. This paper is a first attempt to do so.

We first discuss background material on spyware, including the various types of spyware programs, their methods of transmission, and their run-time behavior. By examining four widespread programs (Gator, Cydoor, SaveNow, and eZula), we present a detailed analysis of their behavior, from which we derive signatures that can be used to detect their presence on remote computers through passive network monitoring. Using these signatures, we quantify the spread of these programs among hosts within the University of Washington by analyzing a week-long trace of network activity. This trace was gathered from August 26th to September 1st, 2003.

From this trace, we show that: (1) these four programs affect approximately 5.1% of active hosts on campus, (2) many computers that contain spyware have more than one spyware program running on them concurrently, and (3) 69% of organizations within the university contain at least one host running spyware. We conclude by discussing security implications of spyware and specific vulnerabilities we found within versions of two of these spyware programs.

1 Introduction

Over the past few years, a relatively new computing phenomenon has gained momentum: the spread of spyware. Although there is no precise definition, the term "spyware" is commonly used to refer to software that, from a user's perspective, gathers information about a computer's use and relays that information back to a third party. This data collection occurs sometimes with, but often without, the knowing consent of the user. In this paper, we use the term spyware in conformity with this common usage.[1] Spyware may also appropriate resources of the computer that it infects [15] or alter the functions of existing applications on the affected computer to the benefit of a third party [12].

Spyware poses several risks. The most conspicuous is compromising a user's privacy by transmitting information about that user's behavior. However, spyware can also detract from the usability and stability of a user's computing environment, and it has the potential to introduce new security vulnerabilities to the infected host. Because spyware is widespread, such vulnerabilities would put millions of computers at risk. In Section 5, we demonstrate vulnerabilities within versions of two widely deployed spyware programs, and we discuss the potential impact of such flaws.

Though most people are aware of spyware, the research community has to date spent little effort understanding the nature and extent of the spyware problem. This paper is an initial attempt to do so. First, we give an overview of spyware in general, in which we discuss the various kinds of spyware programs, their behavior, how they typically infect computers, and the proliferation of new varieties of spyware programs. Next, we examine four particularly widespread spyware programs (Gator, Cydoor, SaveNow, and eZula), and we present a detailed description of their behavior. Our examination was limited to software versions released between August 2003 and the January 2004; as such, our observations and results might not hold for other versions.

Based on our examination, we derive network signatures that can be used to detect the presence of these programs on remote computers by monitoring network traffic. With these signatures, we gather a week-long trace of network traffic exchanged between the University of Washington (a large public university) and the Internet,

[1]Deciding whether a particular program should be called spyware or not can be both difficult and delicate. In practice, there is a continuous spectrum of program behavior that spans from malicious and invasive to fully legitimate. In this paper, we use the term spyware very broadly, and in general apply the term as might be commensurate with the experience of an unsophisticated user. However, we are careful to describe the precise behavior of individual programs discussed in this paper.

from August 26th to September 1st, 2003. We perform a quantitative study of spyware based on this trace, characterizing the spread of the four spyware programs within the university.

Though hundreds of spyware programs exist, our findings show that these four programs alone affect approximately 5.1% of active university hosts, and that these hosts often have more than one spyware program running. Additionally, we find that a majority of organizations within the university contain at least one spyware-infected host, suggesting that existing organization-specific security policies and mechanisms (such as perimeter firewalls) are not effective at preventing spyware installation. Even though our measurements are gathered at only one site, and hence may not be representative of the Internet at large, we believe our results confirm that spyware is a significant problem.

The rest of this paper is organized as follows. In Section 2 we set the context of our study with a brief discussion on the general characteristics of spyware. In Section 3 we narrow our focus to four prevalent spyware programs, giving a detailed description of their behavior. Section 4 presents quantitative results based on our week-long network trace. We discuss implications of our results in Section 5, we present related work in Section 6, and we conclude in Section 7.

2 A Brief Spyware Primer

Spyware exists because information has value. For example, information gathered about the demographics and behavior of Internet users has value to advertisers, the ability to show advertisements correlated with user behavior has value to product vendors, and gathering keystrokes or introducing backdoor vulnerabilities on a host has value to attackers. As long as this value exists, there will be incentive to create spyware programs to capitalize on it.

People are typically exposed to spyware as a result of their behavior. Users may install popular software packages that contain embedded spyware, Web sites may prompt users to install Web browser extensions that contain spyware, and Web browsers retain 'cookies' to track user behavior across collections of cooperating Web sites. The constant growth in the number of Internet users and the increasing amount of time users spend on the Internet have served to amplify users' exposure to spyware.

Spyware succeeds because today's desktop operating systems make spyware simple to build and install. Operating systems and applications are designed to be extensible, and as a result, there are numerous interfaces for interposing on events and interacting with other programs. Operating systems also tend to hide information about background activities to shield users from unwanted complexity. The combination of these two properties makes it difficult to prevent spyware programs from gathering the information they want, or for the user to detect when such information is being harvested or transmitted. As is often the case, there is a tension between usability and security, and to date market pressures appear to favor usability.

2.1 Classes of Spyware

There are many different kinds of spyware. Borrowing from the terminology used in SpyBot S&D [17], a free spyware removal tool, we define the following classes:

- **Cookies and Web bugs:** Cookies are small pieces of state stored on individual clients' Web browsers on behalf of Web servers. Cookies can only be retrieved by the Web site that initially stored them. However, because many sites use the same advertisement provider, these providers can potentially track the behavior of users across many Web sites. Web bugs – invisible images embedded on pages – are related to cookies in that advertisement networks often contract with Web sites to place such bugs on their pages. Cookies and Web bugs are purely passive forms of spyware; they contain no code of their own, relying instead on existing Web browser functions.

- **Browser hijackers:** Hijackers attempt to change a user's Web browser settings to modify their start page, search functionality, or other browser settings. Hijackers, which predominantly affect Windows operating systems, may use one of several mechanisms to achieve their goal: installing a browser extension (called a "browser helper object," or BHO), modifying Windows registry entries, or directly modifying or replacing browser preference files.

- **Keyloggers:** Keyloggers were originally designed to record all keystrokes of users in order to find passwords, credit card numbers, and other sensitive information. Keyloggers have expanded in scope, capturing logs of Websites visited, instant messaging sessions, windows opened, and programs executed.

- **Tracks:** A "track" is a generic name for information recorded by an operating system or application about actions the user has performed. Examples of tracks include recently visited Website lists maintained by most browsers and lists of recently opened files and programs maintained by most operating systems. Although a track is typically innocuous on its own, tracks can be mined by malicious programs.

- **Malware:** Malware refers to a variety of malicious software, including viruses, worms, trojan horses, and automatic phone dialers (which attempt to dial modems to connect to expensive services).

- **Spybots:** Spybots are the prototypical example of "spyware." A spybot monitors a user's behavior, collecting logs of activity and transmitting them to third parties. Examples of collected information include fields typed in Web forms, lists of email addresses to be harvested as spam targets, and lists of visited URLs. A spybot may be installed as a browser helper object, it may exist as a DLL on the host computer, or it may run as a separate process launched whenever the host OS boots.

- **Adware:** "Adware," a more benign variety of spybot, is software that displays advertisements tuned to the user's current activity, potentially reporting aggregate or anonymized browsing behavior to a third party.

Many instances of spyware have the ability to *self-update*, or download new versions of themselves automatically. Self-updating allows spyware authors to introduce new functions over time, but it also may be used to evade anti-spyware tools, by avoiding specific signatures contained within the tools' signature databases.

2.2 The Diversity and Extent of Spyware

Our measurements in Section 4 provide quantitative data on the spread of spyware within an organization. We can also obtain some insight into the extent of the spyware problem by considering other sources of data. One such source of data is the set of spyware signatures that anti-spyware tools have accumulated over time. These signatures are used to compare files and registry entries on a given computer against a list of known spyware programs.

SpyBot S&D [17] is a popular shareware spyware removal tool for Windows-based operating systems. As of January 27, 2004, SpyBot's database contains entries describing 790 different spyware instances. Table 1 breaks down these entries across the previously defined categories. While SpyBot S&D's database is almost certainly incomplete, it demonstrates that there is a substantial number of spyware programs in existence today.

Many spyware infections occur because of spyware programs that are piggybacked on popular software packages. Given this, another interesting source of data to consider is popular shareware and freeware programs. C|Net's *http://download.com/* Website provides free access to over 30,000 freeware and shareware software titles, as well as download statistics about these titles. As a simple experiment, we downloaded the top ten most popular software titles (as of August 2003) and used SpyBot S&D to test each program for spyware.

spyware category	cookies and web bugs	browser hijackers	key-loggers	tracks	malware	spybots
# of DB entries	34	153	62	231	168	142

Table 1. Number of entries in SpyBot S&D's database. The database contains 790 total spyware instances as of January 27, 2004. There is significant diversity in spyware, as these instances are spread across all categories.

Together, these ten titles account for over 872 million reported downloads from the C|Net site. They include three peer-to-peer file-sharing clients, three instant messaging clients, a file compression utility, a download manager, and two anti-spyware tools. Of these ten titles, we found that spyware is packaged with four of them: the software ranked #1, #4, #9, and #10 (Kazaa, iMesh, Morpheus, and Download Accelerator Plus, respectively). These programs have been downloaded over 470 million times. The most popular program (Kazaa Media Desktop) by itself has been downloaded over 265 million times and contains several different types of spyware.[2] Assuming C|Net's data is correct, hundreds of millions of users have been exposed to spyware from this source alone.

To help understand whether the bundling of spyware in free software is a recent phenomenon, we examined several versions of Kazaa Media Desktop released over the past two years. Table 2 shows our results. Twelve different spyware programs have been bundled with Kazaa, and every version of Kazaa released has included at least two different spyware programs. Spyware in free software is not a recent phenomenon—it has been occurring for several years.

Although neither the SpyBot S&D database metrics nor the *http://download.com/* statistics are precise indicators of the extent of the spyware problem, they do reveal that the problem is significant in scope. In the next section of this paper, we narrow our focus to four specific spyware programs. In Section 4, we use our findings to measure the extent to which these four spyware programs have infected hosts at the University of Washington.

3 Gator, Cydoor, SaveNow and eZula

An exhaustive measurement of all types and instances of spyware is well beyond the scope of one paper. Instead, we selected four specific Windows-based programs to examine in detail: Gator, Cydoor, SaveNow,

[2]Kazaa is now distributed in two versions: a free version that contains spyware, and a paid version without spyware.

version	1.3.3	1.4	1.5	1.6	1.7	2.0	2.1	2.1.1	2.6
released	12/01	01/02	02/02	04/02	05/02	09/02	02/03	05/03	11/03
Gator									X
SaveNow	X	X	X	X	X	X	X	X	
Cydoor	X	X	X	X	X	X			X
BDE	X	X	X	X	X	X			
VX2	X	X							
New.net	X	X	X	X	X	X			
OnFlow	X	X						X	
D/L-Ware					X	X	X		
CmnName	X	X	X	X	X	X			X
PromulGate						X			
DirecTVIcon			X	X					
MySearch									X

Table 2. Spyware bundled with Kazaa. This table shows the 10 different programs that were bundled with Kazaa at various points in time, for software versions released between December 2001 and November 2003.

and eZula. We chose these four programs out of the hundreds of possibilities available to us for several reasons. First, anecdotal evidence suggests that these are among the most widely spread instances of spyware. Second, we were successful in deriving signatures that allowed us to detect them remotely with high confidence by sniffing network traffic. Finally, all four are "spybot" or "adware" class programs, according to the classification in the previous section. Because such programs are typically packaged with popular free software, it is particularly easy for an unwitting user to unknowingly install them. For each of the four programs, we give an overview of how they function and what kinds of information they collect.

These four spyware programs each send and retrieve information from remote servers using the HTTP protocol. Because of this, we were able to derive signatures that detect and identify spyware programs operating on remote computers using passive network monitoring. Our signatures are based on two components: lists of servers that each spyware program could potentially communicate with, and HTTP signatures that distinguish spyware activity from human-generated Web browsing activity to those servers. For us to classify a Web request as originating from a particular spyware program, the web request must go to a server associated with that program, and the request must match the HTTP signature associated with that program.

To construct the list of servers with which the spyware programs communicate, we identified all external servers whose DNS name belongs to one of the spyware companies' domain names, or whose IP address belongs to an address prefix allocated to the spyware company according to the ARIN and RIPE registries. Our lists of DNS names associated with spyware servers have 44 entries for Gator, 18 for Cydoor, 12 for SaveNow and 2 for eZula. Our lists of IP address prefixes associated with spyware servers have 4 prefixes for Gator, 9 for Cydoor, 2 for SaveNow, and 1 for eZula. Appendix A presents the server lists and HTTP signatures we used for these programs in full detail.

3.1 Gator

Gator is adware that collects and transmits information about a user's Web activity. Its goal is to gather demographic information and generate a profile of the user's interests for targeted advertisements. Gator may log and transmit URLs that the user visits, partially identifying information such as the user's first name and zip code, and information about the configuration and installed software on the user's machine. Gator also tracks the sites that a user visits, so that it can display its targeted ads at the moment that specific words appear on the user's screen. Gator is also known as OfferCompanion, Trickler, or GAIN.

Gator can be installed on a user's computer in several ways. When a user installs one of several free software programs produced by Claria Corporation (the company that produces Gator), such as a free calendar application or a time synchronization client, the application installs Gator as well. Several peer-to-peer file-sharing clients, such as iMesh [8], Grokster [7], or Kazaa [9], are bundled with Gator. When visited, some Web sites will pop up advertisements on the client's browser that prompt the user to download software that contains Gator. Gator can run either as a DLL linked with the free software that carries it, or within a process of its own launched from an executable called *gain.exe* or *cmesys.exe*. Gator is capable of self-updating.

A rudimentary mechanism to "de-fang" spyware is to remap the DNS names of the spyware servers by adding entries to the client's *hosts.txt* file. By doing so, communication between the spyware client and server is disrupted. However, we observed that Gator inspects the *hosts.txt* file every time the client's computer is rebooted, and comments out any entries that refer to the gator.com domain. Additionally, Gator caches the IP addresses of gator.com DNS names, making it immune to further changes to *hosts.txt*.

3.2 Cydoor

Cydoor displays targeted pop-up advertisements whose contents are dictated by the user's browsing history. When a user is connected to the Internet, the Cydoor client prefetches advertisements from the Cydoor servers. These advertisements are displayed whenever the user runs an application that contains Cydoor, whether the user is online or offline. In addition, Cydoor collects information about certain Web sites that a user visits and periodically uploads this data to its central servers. When a user first installs a program that contains Cydoor, the user is prompted to fill out a demographic questionnaire, the contents of which is transmitted to the Cydoor servers.

Cydoor Technologies (the company that produces Cydoor software) offers a freely downloadable Software

Development Kit (SDK) that can be used to embed the Cydoor DLL in any Windows program, potentially generating advertisement revenue for the program's author. Removing the Cydoor DLL can cause the program that contains it to break.

3.3 SaveNow

SaveNow monitors the Web browsing habits of a user and triggers the display of advertisements when the user appears to be shopping for certain products. While SaveNow does not appear to transmit information about the user's behavior, it does use collected information to target its advertisements. SaveNow will periodically contact external servers in order to update its cached advertisements and its triggers, and to update the executable image itself (*save.exe*). Today's most popular peer-to-peer file-sharing application, Kazaa, is bundled with SaveNow.

3.4 eZula

eZula attaches itself to a client's Web browser and modifies incoming HTML to create links to advertisers from specific keywords. When a client is infected with eZula, these artificial links are displayed and highlighted within rendered HTML. It has been reported that eZula can modify existing HTML links to redirect them to its own advertisers [21], but we have not observed this ourselves. eZula is also known as TopText, ContextPro or HotText. eZula is bundled with several popular file-sharing applications (such as Kazaa and LimeWire), and it can also be downloaded as a standalone tool. eZula runs as a separate process (*ezulamain.exe*) and it includes the ability to self-update.

3.5 Summary

Gator, Cydoor, SaveNow, and eZula vary significantly in functionality, infection mechanism, and the degree of risk they represent to affected users. Through a manual examination of these four programs, we characterized how they operate and we derived HTTP signatures (presented in Appendix A) that can be used to remotely detect infected hosts using passive network monitoring. In the next section of this paper, we use these signatures to measure the activity of spyware-infected hosts within the University of Washington.

4 Measurement and Analysis of Spyware

In this section, we present measurements and analysis of spyware activity at the University of Washington, a large public university with over 60,000 faculty, students and staff, gathered using a week-long passive network trace. We have two main goals: (1) to understand how widespread spyware is within the university, both at the granularity of individual clients and at the granularity of organizations (such as academic departments), and (2) to gain some insight into what kinds of user behavior are correlated with spyware.

	WWW	Gator	Cydoor	SaveNow	eZula
HTTP transactions	120,593,877	489,934	33,122	4,645	5,096
# of clients	31,303	1,077	399	406	63
# of servers contacted	989,794	67	22	3	2
# of orgs. observed	239	154	72	116	40
total bytes transferred	0.95 TB	0.80 GB	149 MB	2.4 MB	37.2 MB
average requests/min	11,964	44.8	3.29	0.46	0.51

Table 3. Trace statistics. Our trace was collected over a week-long period starting on August 26, 2003. "Organizations" refer to groups such as the Department of Physics.

4.1 Methodology

The University of Washington connects to its Internet Service Providers via two border routers. These two routers are connected to four gigabit Ethernet switches, each of which connects to one of four campus backbone links. The switches mirror both incoming and outgoing packets to our passive monitoring host over dedicated gigabit links. Peak bandwidth exchanged between the university and its ISP can reach approximately 800 Mb/s, though the average bandwidth we observed during the trace was 238 Mb/s. The campus contains between 40,000 and 50,000 hosts. Over the period of our trace, we observed 34,983 university IP that exchanged HTTP traffic with external hosts.

The monitoring host reconstructs TCP and HTTP streams from the mirrored packets and produces a log of HTTP activity. Both HTTP requests and HTTP responses are reconstructed; we use heuristics to reconstruct pipelined HTTP requests on persistent connections. All sensitive information, including IP addresses and URLs, is anonymized using keyed one-way hashing before being written to a log. Rather than recording the entire HTTP transaction in the log, our software extracts relavant features (such as source and destination IP addresses, URLs, and transfer lengths) from each request and records them in the log. Hardware counters on the mirroring switches reported 0.000616% packet drops, and the network interface card of the monitoring host showed no packet drops during our measurement interval. Software counters within the kernel packet filter of the monitoring host also reported no packet drops.

In order to preserve locality in anonymized IP addresses, we anonymize each octet of campus IP addresses separately. However, we also zeroed out the last two bits in the last octet of each campus IP address to make the anonymization more secure. We do record the organizational membership (such as individual academic departments and campus dormitories) for each anonymized IP address in the log. Throughout this paper, we identify clients and servers by their IP addresses; this has limitations which we will discuss below.

Our monitoring software classifies each HTTP request before anonymizing and logging it to disk. Any HTTP request to port 80, 8000, or 8080 is classified as WWW traffic. We use our previously derived HTTP signatures to identify traffic from Gator, Cydoor, SaveNow, and eZula within the set of all WWW requests. Finally, as a point of comparison, we identify Kazaa file-sharing transfers by looking for Kazaa-specific headers within all HTTP requests, regardless of the port at which they are directed.

Although our tracing software records all HTTP requests and responses flowing both in and out of our university, the data presented in this paper only considers HTTP requests generated from clients inside the university and the corresponding HTTP responses generated by servers outside the university. Our week-long trace was initiated on August 26, 2003. This trace period corresponds to summer break within campus, so we observed less traffic than when classes are in full session. Table 3 shows a summary of our trace statistics.

4.1.1 Assumptions and Limitations

Our methodology has several inherent limitations. Because we destroy two bits in each IP address during anonymization, we cannot uniquely identify an individual client by IP address alone: each anonymized IP address that appears in the trace log could correspond to four actual IP addresses. However, while collecting the trace, our software maintained counters of the correct number of unique non-anonymized IP addresses in each of the traffic categories in Table 3. Using these counters, we calculated the ratio of the correct number of non-anonymized IP addresses observed by our software to the number of anonymized clients appearing in our log for each traffic category. Whenever needed, we use these "IP address calibration ratios" to back-infer the correct number of IP addresses in a population subset. These ratios are 1.58, 1.05, 1.0, 1.0, and 1.0 for WWW, Gator, Cydoor, SaveNow, and eZula, respectively. All IP-address based population statistics presented in this paper are calibrated using this method.

Because DHCP is used to assign IP addresses in portions of our campus, our methodology of identifying clients by IP address is problematic, as over a sufficiently long time scale, many clients may share the same IP address and an individual client may use several different IP addresses. This "DHCP effect" has been noted in previous studies [3, 11]. To minimize the effects of DHCP, we chose to restrict our trace to a short period of time: one week. Additionally, we excluded the university's dial-up modem pools from our trace, since DHCP issues are particularly problematic for this subset of the university. As we will describe in Section 4.2.1, we were able to calculate the "true" number of Gator clients within the trace, regardless of the anonymization and DHCP issues, using a unique identifier that Gator happens to provide in some of its request headers.

Of the 1,027 active (anonymized) Gator IP addresses observed, we were able to observe Gator identifiers sent from 914 of them. The remainder of the Gator IP addresses did not exchange messages that happened to contain an identifier. From these 914 IP addresses, we counted 872 unique Gator identifiers. The "true" number of Gator clients (872) was therefore inflated by anonymization and DHCP effects to 914, a factor of 1.05. In the rest of this paper, if we quote a population size that is derived from counting Gator identifiers (as opposed to IP addresses), we will explicitly say so.

Another potential problem is that our spyware HTTP signatures could potentially miss some spyware traffic. The signatures distinguish normal user-generated HTTP requests to spyware companies' servers from spyware-generated requests using HTTP patterns. If our signatures happen to miss rarely occurring patterns, then we will underestimate the amount of spyware traffic and potentially the number of spyware-infected clients. Similarly, our spyware signatures filter out traffic based on our list of spyware servers; if this list is incomplete, then we will fail to detect additional spyware traffic. Given that we are only detecting the presence of four specific spyware programs out of the many hundreds that exist, our reported numbers should be considered as a conservative lower bound on the true impact of spyware within the university.

4.2 A Client View of Spyware

We begin by looking at how spyware has affected individual clients within the university. More specifically, we quantify the number of clients with spyware, the rate at which new installations occur, and correlations between various kinds of network activity and susceptibility to spyware installation.

4.2.1 The Spread of Spyware

Over the course of the week, 31,303 internal Web clients accessed 989,794 external Web servers (Table 3). A significant fraction of them, 3.4% (1,077 clients), had Gator installed, 1.3% had Cydoor installed, 1.3% had

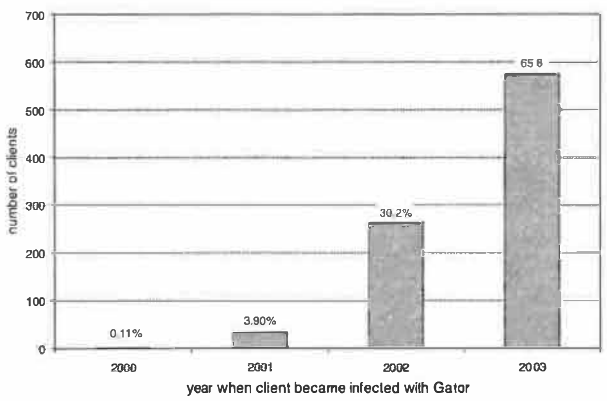


Figure 1. Year of infection for Gator clients. This graph shows the number of currently infected clients that became infected during each of the past three years.

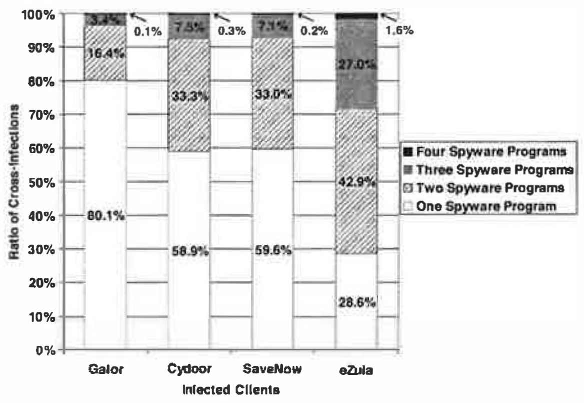


Figure 2. Multiple spyware infections among clients. For each of Gator, Cydoor, SaveNow, and eZula, this chart shows the fraction of infected clients that are infected with exactly one spyware program, two spyware programs, three spyware programs, and all four spyware programs.

SaveNow installed, and 0.2% had eZula installed. *In total, we found that 1,587 clients (5.1%) were infected with one or more spyware programs.*[3]

Although the fraction of active clients that are infected with these four spyware programs may appear to be small, we believe this number is disturbingly high, especially considering that we only measured four out of the hundreds of spyware programs that exist. For example, if a remote exploit exists in Gator, 3.4% of active clients in our university would be susceptible.

During our analysis, we discovered that when Gator is first installed on a client, it performs a series of distinct HTTP requests to "register" the Gator client with the Gator infrastructure. By monitoring these requests, we were able to measure the rate of new Gator installations within the university. Over the course of a week, we detected 52 new installations. Given this slow rate of infection and the fact that we detected 1,077 Gator clients, we hypothesize that many current Gator clients had Gator installed several months or years in the past.

Another fortuitous discovery allowed us to confirm this hypothesis. Many of the messages sent by a Gator client to Gator's central servers carry a timestamp that specifies the precise date of the initial installation. We confirmed that this timestamp survives Gator self-updates. We were able to discover the initial installation date for 872 out of the 1,077 Gator clients; the remaining clients never exchanged a message containing the timestamp. We also used this timestamp to uniquely identify Gator clients within our trace, as mentioned previously in Section 4.1.1.

Figure 1 shows the year of installation for these 872 clients. Over half (65.8%) of clients were installed in 2003, and approximately one third (30.2%) were installed in 2002. Gator has been present within our university for over three years, and one client that was infected in 2000 still remains infected today. Note that these numbers only indicate the number of Gator installations that are still observable today: they do *not* indicate the total number of Gator installations that happened each year. It is possible that other Gator clients exist but either left the university or removed Gator.

4.2.2 Modems Vs. Non-Modems

It is reasonable to expect that spyware will affect personally-owned computers more than university-owned computers, since people have greater freedom to install software on their own machine. To explore whether this is true, we measured the number of infections within the university modem pool to compare against the previously reported statistics for (non-modem) university hosts. Many people dial into the university modem pool from their personal machines. Since DHCP issues are especially problematic for modem pools, we focused on Gator, relying on Gator timestamps rather than IP addresses as unique identifiers within the modem pool.

As previously mentioned, we observed 872 Gator installations within 31,303 non-modem pool university hosts, counting using Gator timestamps. We observed 942 Gator installations in our modem pool (20 of which also appeared in the non-modem pool hosts) within the 12,435 accounts that logged into the modem pool at least once during our trace. Based on these timestamp statistics, 2.8% of non-modem pool hosts were infected with Gator, whereas 7.6% of modem-pool hosts were infected with Gator. Spyware does appear to be more prevalent on personally-owned computers, but it also has a significant presence on university-owned computers.

[3]Note that we could not measure spyware installations on computers that were inactive during our tracing period, so this number is conservative.

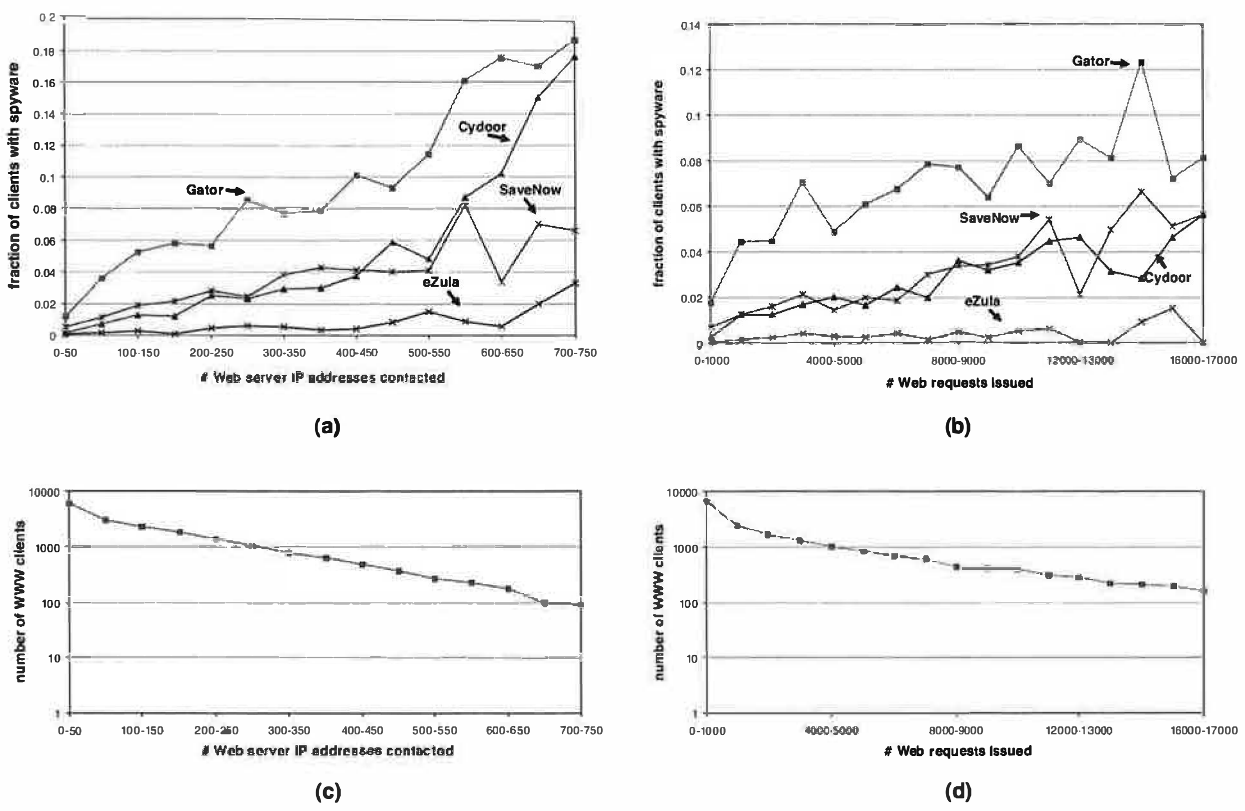


Figure 3. Spyware infections as a function of Web activity. (a) The fraction of WWW clients infected with at least one spyware program as a function of the number of external Web server IP addresses contacted during the week-long trace. (b) The fraction of WWW clients infected with spyware as a function of the number of Web requests issued during the trace. (c) The number of WWW clients (infected or not) as a function of external Web server IP addresses contacted. For example, the point (200-250,1397) shows that there were 1397 Web clients that contacted at least 200 Web servers, but less than 250 Web servers. (d) The number of WWW clients (infected or not) as a function of Web requests issued.

4.2.3 Cross-Infection Rates

The data presented above showed that there were 1,945 spyware infections within the traced (non-modem) population, but only 1,587 computers were infected with spyware. Therefore, many clients must be infected with more than one spyware program. Figure 2 shows, for each of the four spyware programs we detected, what fraction of clients were also infected with other spyware programs. For example, consider the set of clients with Gator. Of these, 80.1% contain only Gator, 16.4% of them contain Gator and one other spyware program, 3.4% of them contain Gator and two other spyware programs, and 0.1% contain all four spyware programs.

In contrast, just 28.6% of eZula clients are infected with only eZula. This suggests that whatever causes eZula infections also causes infections of other spyware programs. Our data shows that many clients infected with spyware are infected with more than one kind of spyware.

4.2.4 Correlating Spyware Infections with Client Behavior

There are many activities that may increase a client's potential exposure to spyware. For example, visiting a large number of Web servers increases a client's likelihood of encountering a spyware-infested Website. As another example, downloading and installing executables off of the Internet may cause a client to unwittingly install spyware. Similarly, installing and running peer-to-peer file-sharing software leads to spyware infections, since file-sharing software often contains bundled spyware. Our trace enables us to examine the correlation between these activities and the fraction of clients with spyware. It is important to note that our results identify correlation, but not necessarily causation.

Ideally, we would restrict our analysis to the behavior of clients at the approximate time when they install spyware. However, since only 52 new Gator installations occurred during the trace, we did not have an adequately large sample of client behavior at the time of new installations. Instead, we considered the behavior of

	number of clients	fraction with Gator	fraction with Cydoor	fraction with SaveNow	fraction with eZula	fraction with any spyware
no EXEs	20,630	0.875%	0.528%	0.776%	0.048%	1.953%
≥1 EXEs	10,673	8.41% [9.6x no EXEs]	2.72% [5.2x no EXEs]	2.31% [3.0x no EXEs]	0.497% [10x no EXEs]	11.1% [5.7x no EXEs]
≥10 EXEs	2,560	10.5% [12xnoEXEs]	5.08% [9.6x no EXEs]	3.09% [4.0x noEXEs]	0.94% [20x no EXEs]	14.7% [7.5x no EXEs]
all	31,303	3.44%	1.28%	1.30%	0.20%	5.07%

Table 4. Spyware infections as a function of downloaded executables. Like Web activity, the number of downloaded executables seems to be correlated with the fraction of clients infected with spyware. Clients that downloaded no executables during the trace had a lower fraction of spyware infections than clients that downloaded multiple executables.

all infected clients, regardless of when the infection took place. This means that our correlations compare activity during the traced time period to infections that may have occurred weeks, months, or years in the past.

Web activity: In Figures 3(a) and (c), we cluster clients into sets according to the number of different Web server IP addresses they contact during our week-long trace. For each cluster shown on the X-axis, we plot the fraction of infected Web clients (Figure 3a) and the total number of Web clients, infected or not (Figure 3c). Figures 3(b) and (d) show similar graphs, except that we cluster clients according to the number of Web requests they issue during the week.

These graphs demonstrate that there is a higher incidence of spyware infections within the clusters of clients with more Web activity. The set of clients that communicated with between 100 and 150 Web servers had a 5.2% Gator infection rate; the set of clients that communicated with 600-650 servers had a 17.5% Gator infection rate. Similarly, the set of clients that issued fewer than 1000 Web requests within the week-long trace had a 1.8% Gator infection rate; those that issued between 12,000 and 13,000 requests had an 8.9% Gator infection rate.

Downloading executables: Downloading executable code from the Internet may expose a client to spyware. Because of this, we expected to see a greater incidence of spyware infections among clients that download many executables.

Table 4 shows the fraction of infected clients as a function of the number of executables they downloaded during our week-long trace. For example, clients that downloaded no executables had a 0.875% Gator infection rate, clients that downloaded 1 or more executables had a 8.41% Gator infection rate, and clients that downloaded 10 or more executables had a 10.5% Gator infection rate. Downloading executables appears to be correlated with higher spyware infection rates.

Using peer-to-peer file-sharing programs: Peer-to-peer file-sharing programs such as Kazaa often ship with bundled spyware. Analysis of our trace revealed that 38% of clients that issued at least one Kazaa request were infected by spyware: 17% of such clients contain Gator, 28.2% contain Cydoor, 8.1% contain SaveNow, and 1.7% contain eZula. These percentages are 5x to 22x times greater than the corresponding infection ratios of Web clients (as reported in Table 3), confirming the intuition that using file-sharing software exposes clients to spyware. However, Kazaa is not the only way clients are exposed to spyware: 62% of clients infected with spyware issued no Kazaa requests during our trace.

4.3 Spyware Bypasses Today's Security Infrastructure

Our university consists of several hundred individual organizations, including academic departments, dormitories, sporting facilities, medical clinics, and many others. Though the core networking infrastructure of the university is centrally managed, each organization is responsible for managing its own end systems and enforcing its own security policies. Some organizations have perimeter firewalls, while others do not. Many organizations centrally manage desktop configurations and are vigilant about installing security patches and anti-virus software updates, while others provide little or no support and have no explicit security policy. Each organization within the university therefore can be considered to be its own independent trust domain, with its own set of defenses against threats and intrusions.

Our network monitor is able to classify network traffic according to these organizational boundaries. Using this classification, we calculated the fraction of organizations that contain one or more spyware-infected hosts. The results are discouraging: 69% of organizations have at least one host infected with at least one variety of spyware. 64% of organizations have Gator infections, 30% have Cydoor, 49% SaveNow, and 17% eZula. Spyware has managed to penetrate most organizations' boundaries, regardless of their security policies. Perimeter protection mechanisms such as firewalls are not helpful, since most spyware infections occur with the cooperation of internal users, whether this cooperating is willingly or unwittingly given.

If a spyware infection leads to a compromise (whether because of a vulnerability in the spyware itself or because of a deliberate backdoor), then an attacker will gain control of a machine inside the organization's trust boundary. As one way of gaining further insight into this issue, we gathered a list of the top one hundred most popular Web servers within the university, ranked according to the number of requests served. Next, we identified which of those Web servers have a Gator client resident on the same /24 subnet (i.e., the IP addresses of the

Web server and the Gator client have the same first three octets). Forty-seven of these Web servers share a subnet with a Gator client, as do two of the top ten. Though some Web servers are isolated from potentially susceptible hosts, many are not.

4.4 Summary

Using passive network monitoring, we have demonstrated that spyware is widespread within the university. More specifically, our results show that:

- at least 5.1% of hosts within the university have been infected with spyware;
- some infected hosts have remained infected for several years;
- many infected hosts contain multiple kinds of spyware;
- the subsets of the university population that use the Web heavily, use peer-to-peer file-sharing software, or download many executable programs tend to have a greater fraction of infected hosts;
- spyware infections span most of the organizations within the university.

The "spyware problem" is significant in scope. In the following section, we discuss security implications of spyware, and we attempt to extrapolate our university-local results to the Internet at large.

5 Discussion

The previous section of the paper demonstrated that spyware is widespread in our university. Spyware can be an inconvenience to infected users, but we argue that it also has significant local and global security implications. As we have shown, spyware exists within most organizations in our university, and therefore has penetrated organizations' security mechanisms. If a widespread spyware program has a vulnerability, then attackers might be able to compromise a significant fraction of machines and penetrate most organizations in the university.

In this section, we describe vulnerabilities that we found in Gator and eZula. Although exploiting them requires the attacker to be able to eavesdrop on and spoof network traffic, it serves to demonstrate that spyware programs do have security weaknesses in practice. After describing the vulnerability, we conservatively estimate how many spyware infections exist within the Internet at large by back-projecting from our university-local results.

5.1 Vulnerabilities in Gator and eZula

Gator and eZula installations consist of both code (e.g., DLLs) and data (e.g., a database of keywords or URLs). Both programs contain self-update mechanisms which allow them to download updates to code or data from a central Website. Upon examination, we found that both Gator and eZula suffered from a simple vulnerability in how they install data file updates.

To update data files, Gator and eZula download compressed archives from their central Websites. The archives are retrieved from URLs that include fully qualified domain names, and therefore the programs issue DNS requests to determine the IP addresses of the Web servers to contact. After downloading a compressed archive, Gator and eZula decompress it and extract the archived files into the local filesystem.

Unfortunately, neither program verifies the authenticity or integrity of a downloaded archive before extracting files from it. Given this, if an attacker can hijack the download TCP connection or spoof *gator.com* or *ezula.com* DNS responses, the attacker can cause a victim to download and install an archive of his choosing. By constructing an archive that contains files with absolute or relative paths in their names, the attacker can place a file in a targeted place within the victim's filesystem. For example, the attacker could place an executable in the "Startup" directory of a user's account by constructing an archive that contains a filename including a path to that directory. While this vulnerability is more difficult to exploit than a buffer overrun vulnerability, it is evidence that spyware programs can and in some cases do contain security flaws. We are not alone in finding such problems: at least one other vulnerability has previously been found in Gator [4].

We implemented and successfully mounted an attack by sending spoofed DNS responses to *gator.com* and *ezula.com* DNS requests coming from infected clients. Our spoofed responses trick the spyware programs into downloading and installing updates that we supply from a local Web server, instead of downloading updates from the intended servers. We verified that we could insert arbitrary executables in our updates, leaving open the possibility of running malicious code or installing backdoors. Though we restricted our testing to victim machines we control, our attack could in practice affect arbitrary machines whose network traffic we can monitor and spoof.

We reported the security vulnerabilities we discovered to the companies that produce Gator and eZula. Claria Corporation (who produces Gator) created an updated version of their software in which the flaw was repaired. To the best of our knowledge, at the time this paper was written, eZula had not yet addressed the vulnerability.

5.2 Estimating the Spread of Spyware on the Internet

Our results only measure the number of spyware clients within our university. Though we expect these numbers to be representative of many organizations besides ours, we wanted to find a way to estimate the extent of the spyware problem on the Internet at large. To do so, we rely on two facts: Kazaa file-sharing software contains embedded spyware (Table 2), and at least 38% of active Kazaa peers within our university are infected with spyware (Section 4.2.4). Using these facts, it seems reasonable to use the presence of Kazaa as an indicator of the presence of spyware.

We have found three different ways to estimate the number of Kazaa installations on the Internet.

- Several Websites maintain counters of the number of active Kazaa users at any given time [16]; these sites generally report that the Kazaa network consists of around 4 million concurrent clients at most times. Using our 38% infection rate, we estimate that there are 1.5 million spyware-infected hosts active on the Kazaa network alone.
- Our measurement infrastructure allows us to identify external Kazaa peers that exchange content with university peers. Using a previously gathered trace, we counted 6,811,743 external Kazaa IP addresses over a 7 month period. This number is likely to be a lower bound on the number of actual Kazaa peers, since only a subset of global Kazaa peers ever contact our university, but using it, we estimate that there are at least 2.6 million spyware-infected Kazaa hosts. Furthermore, these external Kazaa IP addresses spanned over 397,000 external /24 subnets.
- At a different university, a similar study [18] captured 9 million distinct external Kazaa IP addresses interacting with internal hosts. Using this as an estimate, there are at least 3.4 million Kazaa hosts infected with spyware.

All of these estimates confirm that the spyware problem is of significant scope in the Internet at large.

6 Related Work

While we know of no other academic studies on spyware, several commercial efforts have attempted to characterize spyware [17], implement spyware detection removal tools using host-based signatures [1, 17], and estimate the spread of spyware within a customer population [14].

Several previous studies quantified the extent of related Internet security threats, such as self-propagating worms. In 2001, Moore et al. [11] found that more than 359,000 computers became infected with the Code-Red worm in less than 14 hours. In 2003, Moore et al. [10] found over 75,000 hosts infected by the Slammer worm. In this paper, we have demonstrated that spyware affects a similarly large number of hosts in the Internet, and that the existence of vulnerabilities within it makes spyware a potential threat of comparable size and scope.

Intrusion detection systems (IDSs) are a commonly used tool for the prevention and detection of Internet security threats. These systems attempt to identify known attacks, either by monitoring network activity in the case of network-based IDSs [13], or by monitoring host activity in the case of host-based IDSs [5]. The techniques developed for intrusion detection systems may be applicable to the problem of identifying spyware infections. The fact that we were able to derive signatures for passively detecting spyware traffic suggests that this problem is tractable.

A related problem to detecting infections is preventing damage from infections. Many code isolation and sandboxing techniques are potentially applicable, including virtual machines [20, 19], resource containers [2], or system-call sandboxes [6].

7 Conclusions

This paper demonstrates that spyware infections are widespread among hosts in the University of Washington. Our results show that the "spyware problem" is of large scope, and as a result, spyware has significant local and global security implications for today's Internet.

After presenting background material on spyware, we analyzed four specific spyware programs (Gator, Cydoor, SaveNow, and eZula), describing how they function and deriving network signatures that can be used to detect infected remote hosts. Using these signatures, we gathered a week-long trace of network activity at our university, and we used this trace to quantify the spread of spyware on campus.

Our results show that spyware infects at least 5.1% of active hosts on campus, and that many computers tend to have more than one spyware program running on them. We also show that 69% of organizations within the university (e.g., academic departments) contain spyware hosts, suggesting that security practices on campus are not effective at preventing spyware infections.

A vulnerability in a widespread spyware program would potentially put a large number of hosts within the university and in the Internet at risk. We discovered and described a specific vulnerability in Gator and eZula: the potential for spyware to cause substantial security problems is real.

7.1 Acknowledgements

We would like to thank Eric Schwimmer, from whom we initially learned of the prevalence of spyware in peer-to-peer file sharing software. We would also like to thank David Richardson, Ville Aikas, Art Dong and the other members of the Computing and Communications organization at the University of Washington for their support. We are grateful for the guidance of Vern Paxson, our shephard, and our anonymous reviewers. We thank Richard Dunn, Krishna Gummadi, David Wetherall, and John Zahorjan for their helpful comments on this research. This research was supported in part by the National Science Foundation under Grants ITR-0121341 and CCR-0085670, and by a gift from Intel Corporation.

References

[1] Ad-Aware. `http://www.lavasoftusa.com/software/adaware/`.

[2] G. Banga, P. Druschel, and J. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI 1999)*, New Orleans, LA, February 1999.

[3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-peer Systems*, Berkeley, CA, December 2002.

[4] EyeOnSecurity. `http://eyeonsecurity.org/advisories/Gator/`.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, Oakland, CA, May 1996.

[6] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, July 1996.

[7] Grokster. `http://www.grokster.com`.

[8] iMesh. `http://www.imesh.com`.

[9] Kazaa. `http://www.kazaa.com`.

[10] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[11] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm,. In *Proceedings of the 2002 ACM SIGCOMM/USENIX Internet Measurement Workshop*, Marseille, France, November 2002.

[12] S. Olsen. Software replaces banner ads on top sites. C|Net News.Com article, August 2001.

[13] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[14] PC Pitstop. PC Pitstop spyware statistics. `http://www.pcpitstop.com/research/spyware.asp`, January 2003.

[15] J. Schartz. "Acquitted man says virus put pornography on computer". New York Times article, August 2003.

[16] Slyck. `http://www.slyck.com`.

[17] SpyBot S&D. `http://security.kolla.de`.

[18] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 2002 USENIX Security Symposium*, San Francisco, CA, August 2002.

[19] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 Annual USENIX Technical Conference*, Boston, MA, USA, June 2001.

[20] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[21] J. Wilson. How TopText works. `http://scumware.com/wm2.html`.

A Spyware Signatures

To identify spyware traffic using our passive network monitor, we derived network signatures for each spyware program we wanted to identify. We classify a web request as originating from spyware if the request matches both of the following signature components:

- **Server list:** the web request must originate from a client within the University of Washington, and be directed at a server in a "server list." For each spyware program, we constructed a list of IP addresses and DNS names associated with servers known to be operated by the spyware program's company.
- **HTTP signature:** the web request must contain some HTTP signature that the spyware program includes in its requests, but which web browsers are unlikely to generate. This signature may be the value of the HTTP "User-Agent" field, or it may be a URL pattern.

In this appendix, we list the signatures we derived for Gator, Cydoor, SaveNow, and eZula.

A.1 Gator

Server list:

```
autoupdate.balance.gator.com
bannerserver.balance.gator.com
bannerserver.gator.com beasley.gator.com
```

```
bg.gator.com content.balance.gator.com
coupons.gator.com dns-a.gator.com
dns-cw.gator.com dns.gator.com
dns2.gator.com gator.com gator29.gator.com
gatorcme.gator.com gi.balance.gator.com
gi.gator.com gs.balance.gator.com
gs.gator.com gw-rwc.gator.com
images.gator.com jeeves.balance.gator.com
map.gator.com outsidedns.gator.com
patchserver.balance.gator.com
pricecomparison.gator.com rs.gator.com
search.balance.gator.com search.gator.com
scriptserver.gator.com ss.balance.gator.com
ss.gator.com ssbackup.balance.gator.com
surveys.balance.gator.com trickle.gator.com
trickle.balance.gator.com ts.gator.com
updateserver.gator.com wb.gator.com
web.balance.gator.com webpdp.gator.com
webpdp.balance.gator.com www.gator.com
xmlsearch.balance.gator.com xmlsearch.gator.com
```

```
63.197.87.0/24 64.94.89.0/24 64.152.73.0/24
66.35.229.0/24
```

HTTP signature: The Gator program uses the custom User-Agent HTTP header "Gator/x.xx", where "x.xx" is the version number of the Gator client.

A.2 Cydoor

Server list:

```
www.bns2.net www.bns1.net
www.rgs1.net www.rgs2.net
www.cms1.net www.cms2.net
cydoor.com www.cydoor.com
globix.alteon.cydoor.com
globix.alteon2.cydoor.com
jcms.cydoor.com jbns.cydoor.com
jbn2.cydoor.com jbns2.cydoor.com
jbnss.cydoor.com jmbns.cydoor.com
jmcms.cydoor.com sprint.alteon1.cydoor.com
```

```
63.170.89.0/24 209.10.17.128/25 209.73.225.0/24
209.11.66.0/24 209.11.84.130/32 209.11.84.135/32
209.11.84.137/32 209.11.84.138/32
209.11.84.139/32
```

HTTP signature: To detect Cydoor, we use specific keywords in the URL of the HTTP requests. In particular, we identify as Cydoor traffic any request to a server in the server list that contains a URL whose prefix is "/bns" or "/scripts," or a URL containing the string "javasite." The "bns" keyword refers to requests for pop-up advertisements. The "scripts" and "javasite" keywords refer to scripts that are used to collect information from users (note that such information is obfuscated but not encrypted).

A.3 SaveNow

Server list:

```
app.whenu.com chromium.whenu.com
iron.whenu.com lead.whenu.com
mercury.whenu.com oxygen.whenu.com
tin.whenu.com titanium.whenu.com
web.whenu.com whenushop.whenu.com
helium.whenu.com zinc.whenu.com
```

```
209.11.45.128/27 209.73.202.0/27
```

HTTP signature: Similar to Cydoor, we use specific keywords in the URLs within requests to SaveNow servers to detect SaveNow client activity. Specifically, we identify as SaveNow traffic any URL whose prefix is "/offer," "/heartbeat," or "/about." The "offer" keyword is usually followed by a list of parameters denoting a Website visited or a keyword entered by user within a form. These are presumably used for determining which advertisement should be displayed to the client. The "heartbeat" keyword is also followed by a list of parameters indicating a name of a program, a "partner code" and an "id code." We assume that this is a heartbeat mechanism for SaveNow that identifies which program is running SaveNow. The "about" keyword also includes a list of strings that also appear to be used to select advertisements that are displayed to the client.

A.4 eZula

Server list:

```
app.ezula.com ezula.com
```

```
208.185.211.64/26
```

HTTP signature: We use the HTTP User-Agent field to identify eZula spyware traffic sent to eZula servers. The eZula spyware program uses three specific User-Agent fields: "eZula," "mez," or "Wise," depending on the specific program version.

Model Checking Large Network Protocol Implementations

Madanlal Musuvathi [*], Dawson R. Engler
{madan, engler}@cs.stanford.edu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A

Abstract

Network protocols must work. The effects of protocol specification or implementation errors range from reduced performance, to security breaches, to bringing down entire networks. However, network protocols are difficult to test due to the exponential size of the state space they define. Ideally, a protocol implementation must be validated against all possible events (packet arrivals, packet losses, timeouts, etc.) in all possible protocol states. Conventional means of testing can explore only a minute fraction of these possible combinations.

This paper focuses on how to effectively find errors in large network protocol implementations using model checking, a formal verification technique. Model checking involves a systematic exploration of the possible states of a system, and is well-suited to finding intricate errors lurking deep in exponential state spaces. Its primary limitation has been the effort needed to use it on software. The primary contribution of this paper are novel techniques that allow us to model check complex, real-world, well-tested protocol implementations with reasonable effort. We have implemented these techniques in CMC, a C model checker [30] and applied the result to the Linux TCP/IP implementation, finding four errors in the protocol implementation.

1 Introduction

Network protocols must work. The current state-of-practice for automatically ensuring they do are various forms of testing — using a network simulator, doing end-to-end tests on a live system, or as an interesting twist, analyzing their traces to find anomalies. The great strength of such testing approaches is that they are easily understood and give an effective, lightweight way to check that the common case of an implementation works. Unfortunately, protocols define an explosively large state space. Implementors must carefully handle all possible events (lost, reordered, duplicated packets) in all possible protocol and network states (with one packet in flight, with two, with a just-wrapped sequence number). It is only possible to test a minute fraction of the exponential number of such combinations. Thus, just at the moment implementors need validation the most, testing works the least well. As a result, even heavily-tested systems can have a residue of errors that take days or even weeks to arise, making them all but impossible to replicate.

When applicable, formal verification methods can find such deep errors [26, 32, 37]. One approach involves an *explicit state* model checker that starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

Most conventional model checkers, however, require that an abstract specification (commonly referred to as the "model") of the system be provided. This upfront cost has traditionally made model checking completely impractical for large systems. A sufficiently detailed model can be as large as the checked system. Thus, implementors often refuse to write them, those that are written have errors and, even if they do not, they "drift" as the implementation is modified but the model is not.

Recent work has developed model checkers that work with the implementation code directly with-

*Supported by GSRC/MARCO Grant No:SA3276JB

out requiring an abstract specification. In prior work, we developed an implementation-level model checker, CMC [30], and used it to check three different implementations of the AODV protocol (roughly 6K lines of code each). Model checking AODV involved extracting the AODV processing code from the implementation and running it along with an input generating test harness. Using this approach, CMC found errors every few hundred lines of code, as well as an error in the underlying AODV protocol specification [12].

This paper is about how to scale model checking up to protocols a factor of ten larger. After the success checking AODV we decided to check the hardest thing we could think of: the Linux kernel's widely-used (and thus extremely thoroughly tested and visually inspected) implementation of TCP. The particular implementation we used (version 2.4.19) is roughly 50K lines of code.

Scaling CMC to such a large system involved some unusual challenges. First and foremost was the "unit-testing" problem — model checking TCP requires running the kernel implementation as a closed system in the context of CMC. However, extracting large pieces of code from a host system not designed for unit testing is much, much harder than it may seem. Any procedure this code calls must be reimplemented in the model checker; real code has a startling number of such procedures (the narrowest interface we could cut along in TCP had over 150 procedures). Worse, such procedures too-often have unspecified interfaces, making it easy to get their functionality slightly wrong. Model checkers are excellent at finding slightly wrong code, and will happily detect the resulting bogus effects, requiring a laborious tracking back to the source of these false errors. In many cases, this backtracking took days.

To avoid these problems, this paper presents an unusual approach; instead of extracting the TCP implementation, we run the *entire* Linux kernel in CMC. To trigger TCP related behaviors, the system contains an environment that interacts with the kernel through well-defined interfaces, such as the system call interface and the hardware abstraction layer. The semantics of these interfaces are well understood and thus, easy to implement correctly. The execution of the TCP code in the model checker exactly mirrors the execution in the kernel, thereby reducing false errors.

Running the entire kernel in CMC required us to heavily optimize the model checker. The system consists of *two* kernels communicating with each other as TCP peers, and the size of the system state is over two hundred kilobytes. This paper describes techniques that enable CMC to scale to such a large system and validates them by applying them to the Linux TCP implementation, where we find four errors. We believe that the approach we took and the techniques we used are useful (and perhaps necessary) to model check real code of any size.

This paper makes the following contributions:

1. It develops novel techniques to check code *in situ* rather than requiring it to be extracted from the system itself.
2. It develops ways for saving and restoring state to be automatic, yet efficient, and for superficial differences at the bit-level to be eliminated.
3. To the best of our knowledge, it is the first paper to check software as complex as TCP; the closest other efforts are an order of magnitude smaller.
4. It demonstrates that the approach can find real error in heavily inspected and tested, complex code.
5. It provides a generic framework for testing other protocols with much lower effort than any other model checker.

2 CMC Overview

CMC is an explicit model checker that directly executes a given protocol implementation. One way to understand the working of CMC is to consider it as a *backtracking* network simulator. Like any network simulator [25], CMC runs a protocol description along with a suitable environment that consists of a network model and a user model. Instead of using a simplified protocol description, CMC executes an actual implementation of the protocol (along the lines of [7]). As in a network simulator, CMC enables protocol behavior by triggering the events in the environment, such as network interrupts, timeouts, and user inputs.

However, the key difference between CMC and a network simulator is that CMC checkpoints the system state at specific points. This provides two important capabilities. First, CMC can backtrack to a previous state and explore a different sequence of events. In contrast, a network simulator is restricted to exploring a single event sequence determined by the initial random seed provided to the simulator. Second, CMC stores a signature of each checkpointed state in a hash table. By doing so, CMC avoids redundantly exploring a system state more than once. This is particularly helpful when exploring all event interleavings in the system.

CMC is implemented as a library that links with the C (or C++) implementation of the network protocol. CMC models a given system as one or more interacting *processes*. Each process can have one or more threads. A CMC process behaves in many respects like a normal operating system process; each process has its own copy of global variables, heap, and stacks for each of its threads. Processes do not share state, but communicate with each other through a region of shared memory. CMC along with all the process in the system run as a *single* operating system process. Internally, each process is implemented as one or more user space threads.

The notions of processes and threads provide a straightforward way to model network protocols. Each process models a node in the network and executes the implementation of the protocol. If the implementation is multithreaded (as is the case with the Linux TCP implementation), a CMC process can allocate a thread for each threads of execution in the implementation. The processes can communicate with each other using a network modeled using the shared memory region.

A user has to perform the following tasks before a protocol can be model checked. First, the protocol implementation should run as a closed system in a CMC process. Typically, this involves extracting the protocol specific parts from an implementation and providing *stubs* or support functions to close the system. This task is necessary for unit-testing the protocol implementation, or to execute the implementation *as is* in a network simulator [7].

Second, the user specifies the *nondeterminism* in the system. This allows CMC to explore multiple executions from a single state. In most cases, this involves calling special functions in the CMC library. For instance, CMC provides a `send()` function that nondeterministically drops any packet sent to the network. As a result, CMC explores two possible executions for every packet; one in which the packet is sent to the network, and the other in which the packet is dropped.

Third, the user provides some correctness properties about the protocol. The user specifies these properties as invariants implemented as boolean functions (in C), which CMC evaluates at each state during state space exploration. For instance, a user can provide a function that checks if a TCP implementation reduces the congestion window in response to a packet loss.

Given an appropriate system that includes the protocol implementation, CMC systematically enumerates the possible states of the system. The system state includes the state of all processes and the state of the shared memory region. The state of process consists of the contents of all the global variables, the heap, and the stacks (and register contents) of all the threads in the process.

CMC starts from the initial state of the system, and recursively generates all its successors. From each state, CMC executes a *transition* to generate a successor state. A transition roughly corresponds to the handling of a protocol event, such a packet reception or a timeout, by some thread of a process in the system. Due to nondeterminism in the system, there can be more than one transition possible from a state, each potentially leading to a different successor state.

The state space is prohibitively large for most nontrivial systems. As a result, it is impossible to enumerate *all* possible states of a given system with limited resources. While not able to provide absolute guarantees about a given protocol implementation, CMC focuses on exploring as many different protocol behaviors as possible before running out of resources (§5).

3 The Model Checking Framework

The first step in model checking a protocol is to run the protocol implementation as a process in CMC. In our case, this requires that the Linux kernel implementation run as a closed system in user space.

3.1 Why the Conventional Approach Fails

In our first attempt, we followed the conventional wisdom: extract the TCP related code from the kernel along the narrowest possible interface; and run it with a *kernel library* that provides stubs for all the kernel services the TCP code requires. Choosing a narrow interface keeps the library simple. This approach also has the advantage of minimizing the state size of the model, as the kernel library can be optimized by removing any redundant states.

Starting from the core set of TCP modules, we conservatively added a few tightly coupled modules (such as IP) to the model. To close the system, we then manually provided stub implementations for all the interface functions in the system boundary.

However, providing correct implementations for these interface functions proved to be an extremely difficult task. The TCP code interacts with the rest of the kernel along complex and undocumented interfaces. Our initial version of the kernel library involved around 150 interface functions. Providing stub implementations and understanding the subtle interactions between various interface functions required considerable understanding of the different kernel modules. More often than not, our stubs were buggy.

Faulty stubs typically result in false behaviors that CMC will (falsely) flag as errors in the checked code. These false positives can be very hard to debug and fix. For instance, after days of debugging we found that a memory leak of a socket structure was caused by incorrect stub implementation in the timer module. The TCP implementation uses a function `mod_timer()` to modify the expiration time of a previously queued timer. This function's return value depends on whether the timer is pending when the function is called. However, our initial stub implementation did not capture this behavior. This incorrect stub confused the reference counting mechanism of the socket structures leading to a memory leak. (As TCP timers are members of the socket structure, a queued timer amounts to an extra reference to the parent socket.)

During our initial runs, we progressively fixed bugs in our implementations as we found them. After spending months, we gave up. It is quite possible that after sufficient iterations of fixing errors in the environment model, we would have converged on a model that implemented all the interfaces accurately. However, subsequent iterations involved bugs that were more subtle and took longer to debug.

3.2 Running the Entire Linux Kernel

The hard learned lesson from our previous approach is that instead of choosing a *narrow* interface, the model should involve *well-defined* interfaces. For the Linux kernel, there are only two such interfaces: the system call interface that defines the interface between the kernel and the user processes; and the "hardware abstraction layer" that defines the interface between the kernel and the hardware architecture. Though Linux does not explicitly define a hardware abstraction interface, such an interface is implicitly defined for most kernels to simplify the task of porting the kernel to different architectures.

Defining the TCP model along these two interfaces requires that the *entire* kernel is run in user space as a CMC process. While this might look like a daunting task, we heavily reused the user space implementation of the Linux kernel from [38]. This still requires CMC to deal with the entire kernel state which is orders of magnitude larger than the TCP relevant state alone. Section 4 describes techniques by which CMC in effect automatically extracts the TCP relevant state from the kernel state.

Using the system call interface and the hardware abstraction interface has another advantage. These interfaces change very rarely during future revisions. Thus, the effort required in building a TCP model can be reused across multiple versions of the kernel.

3.3 The TCP Model

Once the TCP implementation can run in user space, the next step involves constructing the actual CMC model: allocating one or more CMC processes to run the TCP implementation and designing an environment to appropriately trigger the implementation.

In the kernel, the TCP code executes in three contexts: in user context when a user process makes a system call, in the context of a network interrupt handler when a TCP packet is received, and in the context of a timer interrupt handler when a TCP timer fires. To mimic this behavior a CMC process running the Linux TCP implementation contains the following three threads:

- An *application* thread that makes socket related system calls to the kernel. Of the two application threads one behaves as a standard TCP server, while the other behaves as a standard TCP client.
- A *network* thread that emulates a packet interrupt and executes the code to handle packet reception.
- A *timer* thread that emulates a timer interrupt and fires one of the pending timer routines.

These threads once triggered can either execute to completion or can block. These threads block by *yielding* control to the kernel scheduler. In the real kernel, the scheduler would then execute the scheduling algorithm to determine the thread that

	Size of a System State (in KB)	**Average Change in a Transition (in KB)**
Global Variables	78.28	11.37
Heap (average)	25.06	2.13
Stack	24.00	4.00
Total per Process	127.34	17.50
Network State	1.00	1.00
System State (Sum of two process states and a network state)	255.68	18.50

Table 1: The state size for the TCP model described in Section 3.3. The second column shows the amount that changes in a transition, on average. For the global variables and the stack, CMC can only detect changes at page size (4KB) granularities. For the heap, CMC detects changes at individual object granularities.

is scheduled next. In our model, the scheduler is modified to immediately transfer control to CMC. This enables CMC to *nondeterministically* choose the thread that is executed next, and explore multiple thread schedules from a given state.

Similarly, the kernel timer routine is modified to allow CMC to choose the timer that fires next when multiple timers are enabled. Optionally, CMC can fire timers out of order irrespective of their expiration times. While this can lead to some behaviors not possible in a real implementation, it has the benefit of making the implementation behavior independent of the actual values of the timers.

The two kernels communicate through a network, modeled as a list of messages in the shared memory. The network model can lose, duplicate, reorder and corrupt messages. Each kernel communicates with the network through an appropriate network device driver.

4 Handling Large States

Protocol implementations can have large states. For the TCP model discussed in Section 3.3, each state is around 250 kilobytes, which is shown in Table 1. Note that a process in the TCP model runs the entire Linux kernel. Thus, the process state consists of all the global variables in the kernel (78KB), and any memory dynamically allocated during the kernel boot-up and the subsequent processing of TCP events (25KB). Additionally, the process runs three kernel threads (§3.3), each of which run in a separate 8KB stack. [1] The system consists of two such processes along with a model of the network, resulting in a state that is 250KB in size.

[1]This stack size is hard-coded in the Linux kernel implementation.

4.1 Managing Memory Resources

During the state space search, CMC maintains two data structures: a hash table of states seen during the search, and a queue of states seen but whose successors are yet to be generated. It is not necessary to store the entire state in a hash table [36]. CMC uses a hash compaction algorithm [36] to store only a small signature (typically, 8 bytes) for each state in the hash table. By doing so, the memory requirements of the hash table depend only on the number of states explored during model checking.

The queue, however, has to maintain the states in their entirety, as all of the information in the state is necessary to generate the successor states. However, the queue has good locality of reference, so much of it can be swapped to disk during model checking. Moreover, the states in the queue can be efficiently compressed; as states can simply be regenerated by remembering the sequence of events that generated them from the initial system state.

In practice, the large amount of memory available in modern machines makes managing memory much easier. For instance, a gigabyte hash table is more than sufficient to explore 100 million states when using a 8 byte compacted signature for each state. The remaining memory can be allocated for the queue. As will be shown below, CMC is limited more by the time required for the state space exploration than by the memory available.

	Time in microseconds				
	State Restore	**Transition Execution**	**Hash Computation**	**State Store**	**Total**
Processing Entire States	656	305	17816	608	19385
With Incremental States	298	363	2927	64	3652
With Incremental Heap Canonicalization	305	365	1453	65	2188

Table 2: Time taken for a single CMC transition, averaged over the first million transitions. The experiment involves CMC checking the TCP model in a server running a 800MHz Intel Pentium III processor with 256KB cache and 2GB of memory.

4.2 Time Taken for a Transition

The basic step of CMC consists of a transition, which generates a single successor from a particular state. A transition consists of the following steps.

1. **State Restoring:** First, CMC restores the system to the desired start state of the transition.

2. **Executing the Transition:** After restoring the state, CMC transfers control to one of the enabled threads in the system. The thread executes the TCP code to process a specific input event such as a packet reception or a timeout.

3. **Computing the Hash:** When the thread yields control back to CMC, the implementation state, as modified by the thread, represents the successor state of the transition. To store this state in the hash table, CMC computes a signature and a hash value (that determines the location of the signature in the hash table) for the state. Additionally, CMC might perform state transformations (§5.1) before this hash computation.

4. **State Storing:** If the successor state is not present in the hash table, CMC queues a copy of the successor state for further exploration.

Thus, each transition requires at least three traversals of the state contents. When the state is hundreds of kilobytes, naively performing these traversals leads to a poor memory cache performance, considerably slowing the model checker. Table 2 shows the time taken for a transition when CMC processes the entire state contents during the transition. In this case, each transition takes around 20 milliseconds. At this rate, CMC can run for weeks without running out of memory. (A gigabyte hash table can easily store 100 million states, and for the TCP model, only one in five transitions generate a new state.)

From Table 2, we can see that the hash computation is the most expensive step in a transition; CMC spends more than 90% of its time computing the signature and the hash value of the state. While saving and restoring states involve simple memory copies, hash computation requires performing a few arithmetic operations for *every* byte of the state.

4.3 Incremental State Processing

It is possible to reduce the hash computation overhead by using a simple, but crucial observation. Even though the state is large, only small portions of it change in a transition, as shown in Table 1. There are a couple of reasons for this behavior. As the environment triggers only TCP specific events, parts of the kernel not relevant to TCP processing do not change during model checking. Additionally, a transition involves a specific event and thus, only involves data structures related to the processing of that event.

By processing only the incremental differences between states, CMC can considerably reduce the time taken for a transition. The basic idea is to subdivide an entire state into a set of smaller *objects*. CMC computes the hash value and the signature for each object separately and caches these values with the object. CMC identifies the objects that are modified in a transition, and only needs to process these objects during hash computations. The cached values can be used for objects not modified in a transition.

To determine the objects modified in a transition, CMC uses the virtual memory protection allowed by the underlying operating system. This scheme is similar to that used by distributed shared memory systems (such as Treadmarks [23]). The entire state is subdivided into a set of virtual memory pages. Before a transition, CMC restores the system to the desired start state and write protects all the pages in the state. During the transition, a first write to a protected page generates an access violation signal. CMC traps this signal, marks the page as dirty, and creates a copy of the page. This copy represents the state of the page in the start state of the transition. Once a page is dirty, CMC ensures that subsequent writes to the page do not involve an expensive signal delivery by removing the write protection on the page.

Table 2 shows the performance improvement in this incremental scheme. The cost of hash computation reduces by more than a factor of 5, and the transition takes less than 4 milliseconds on average. There is also an improvement in the state store and restore phase, as CMC only needs to copy pages that differ when switching between states. As expected, the time for actually running the implementation code increases due to the overhead of the access violation signals.

Section 5.1 describes another mechanism to further reduce the hash computation overhead.

5 Handling State Space Explosion

All model checkers have to handle the *state explosion* problem, which refers to the unmanageable size of state spaces even for moderately sized systems. As CMC deals directly with protocol implementations rather than their abstract models, CMC confronts much larger state spaces than conventional model checkers.

CMC tackles the state space explosion problem by following a best-efforts approach. CMC does not guarantee a complete search of the state space, but attempts to explore as many protocol behaviors as possible before running out of resources. No current approach can practically verify protocols of any complexity, so we instead focus on exercising the protocol as throughly as possible.

There are two ways in which CMC confronts the state explosion problem. First, CMC (like all model checkers) uses state transformations to recognize states that are superficially different at the bit level but are actually the same (or similar enough) at the semantic level. These transformations enable CMC to check only one out of a large (and exponential) set of equivalent states. Second, CMC uses heuristics to selective explore interesting portions of the state space.

This section describes two techniques that are an improvement of those previously described in [30]. Scaling CMC to TCP required that these techniques be automated and made more efficient.

5.1 Incremental Heap Canonicalization

One problem in model checking software programs is to handle heap canonicalization [22]. When objects are dynamically allocated in the heap, different allocation orders can result in heap states that are equivalent but differ in the memory locations of the objects. For instance, consider two states in which the TCP implementation receives the same two data packets in order and out of order respectively. The socket buffers will be allocated at different memory locations in the two states but queued in the sequence number order in both states. Thus, the two states differ in the bit level but are semantically the same. CMC should identify such states and avoid exploring them redundantly.

Equivalent heap states can be identified by transforming a heap state to a canonical representation shared by all equivalent heaps. Informally, the objects in the heap along with their pointers form a heap graph. Equivalent heaps have the same underlying graph structure. A canonicalization algorithm works by relocating each object to its canonical location and modifying the contents of all pointers to the object to reflect its new location.

The previously known algorithm [22] does not scale to large heaps. This algorithm requires processing large portions of the heap, which can reduce the speed of state space exploration (§4.2). Specifically, it performs a depth first traversal of the heap graph, and the canonical location of each object depends on its depth first ordering. Even small changes to the heap, such as an object allocation or a deletion, can change the depth first ordering for a large number of objects. This forces CMC to traverse and compute the hash for large portions of the heap, as shown in Table 3. During model checking, the heap, on average, consists of 103 objects that total 25KB. Note that the heap also includes non-TCP related data structures allocated by the Linux kernel. A TCP transition, on an average modifies 5 of these objects. However, this change requires CMC to recompute the hash value of almost half the objects in the heap.

Our contribution is an improved, *incremental* heap canonicalization algorithm. We briefly describe this algorithm; interested readers can refer [28] for more details. The incremental algorithm generates the

	Number of Objects	Total Length in KB
Objects in the entire heap	102.7	25.06
Objects modified in a transition	5.1	2.14
Objects accessed during heap canon.	45.8	11.91
Objects accessed during incremental heap canon.	5.2	2.17

Table 3: Objects accessed during heap canonicalization. Every transition on an average modifies 5 heap objects. The incremental canonicalization algorithm requires traversing only these objects in most cases.

canonical location of a heap object from the *shortest path* of the object to some global variable in the heap graph. When a transition makes small changes to the heap structure, the shortest path of most objects is likely to remain the same [19, 31]. After a transition, CMC recomputes the hash only for those objects whose shortest paths have changed in a transition. This algorithm works well in practice, as shown in Table 3; in most cases CMC traverses only the objects that are modified in a transition. This improves the performance of CMC by 40% as indicated in Table 2.

5.2 Exploring Interesting Protocol Behaviors

Since CMC cannot exhaustively explore the state space of real protocols it takes a different approach and instead tries to explore the "most interesting" portions of them. It does so by attempting to focus on states that are the most different from previously explored states. The intuition for this is that the more different a state is from previous states the more likely it is to have new behaviors and, as a result, bugs. We describe two techniques CMC uses below.

The first uses the abstract protocol state to find states that will generate new behaviors. Conceptually, a protocol implementation state can be viewed at two levels. There is the concrete state, which is the heap, stack, global variables and registers — all the memory values that define the current computation. There is also the abstract, protocol state encoded in the concrete state — for example, whether TCP is in the LISTEN or CLOSED state. Many different concrete states may in fact define the same abstract state. We want to focus on concrete states that correspond to new abstract protocol states since they will generate new behaviors.

For TCP, the state of the reference model (§6) corresponds to the the abstract state of the protocol. We can thus use it to preferentially explore system states whose corresponding reference model state has not been seen before. We further tune this process by doing a series of *symmetry reductions* [10] on the abstract state, which allow us to determine when two superficially different abstract states will in fact generate the same behavior. The simplest example is sequence number standardization where, a state with all sequence numbers at 256 can be considered equivalent to a state with all sequences numbers at 512 (which can be reached, for instance, from the first state after successful data transfer of 256 bytes). While performing such reductions on the concrete state is difficult the small size of the abstract state ($10 - 20$ bytes) makes it relatively easy.

The second technique also explores states that appear to be different from previous ones. It uses the heuristic that on balance, a change in a variable that never or rarely changed before is more interesting than in one that changes often. CMC tracks the values of all variables in the states generated. If a particular variable (as determined by its byte location in the state) takes on more than a threshold number of distinct values, CMC eliminates those variables from subsequent hash computations. Different CMC runs can use different threshold values (including infinity) to guard against cutting off searches too soon.

A simple example of how this helps are the many counters and and statistics variables present in protocol implementations that do not affect the execution of the protocol, but whose changed values cause unnecessary blowup of the state space. Ideally, a user would identify such variables and provide them to CMC as annotations. However, providing such manual annotations for the entire TCP implementation (and in our case, the entire Linux Kernel) is impractical. CMC's variable pruning will automatically weed out such counter variables.

6 Specifying Correctness Properties

During the state space exploration, CMC automatically checks for certain generic properties such as memory leaks and invalid memory accesses. Also, CMC reports any deadlock states, in which the system can make no progress. To check for protocol-specific properties, the user has to provide additional invariants (written in C). CMC evaluates these invariants in every state it generates.

There are two aspects to the correctness of a network protocol. First, the protocol specification should be correct. Second, the particular implementation should implement the specification correctly. By running the implementation, CMC can simultaneously check for both specification and implementation errors. Any specification error will be promptly represented in the implementation. Given the maturity of the TCP specification, however, it is quite unlikely the specification contains errors and our emphasis has been on detecting implementation errors.

6.1 Checking Protocol Conformance

Checking that a TCP implementation conforms to the protocol specification is a challenge. The specification itself [35, 6] is large and complex. Moreover, the TCP specification is ambiguous at many places, leaving room for the implementations to make their own choices.

Along the lines of [33], we check for protocol conformance by ensuring that every transition of the implementation is allowed by a TCP reference model. This reference model consists of the basic state machine transitions literally translated from [35] to C, and is around 500 lines of code. During model checking, CMC provides the same set of input events (system calls, network and timer interrupts) to both the implementation and the reference model, and expects their states and the outputs (network packets and system call return values) to be consistent.

All inconsistencies between the TCP implementation and the reference model are not necessarily errors. They can arise due to the ambiguities in the TCP specification, known errors in the protocol [34], and manual errors in the reference model. We iteratively modified our reference model when we found such false error reports.

6.2 Checking for Resource Leaks

A TCP implementation should release all kernel resources at the end of the connection. Failure to do so can result in resource lockup, which subsequently reduces the performance and the availability of the machine. The resource leaks are not necessarily memory leaks, as these resources can still have valid references to them. To check for such resource leaks, CMC requires that the entire kernel eventually reach the initial state after completing a TCP connection. CMC reports any violation as a potential resource leak.

There are, however, valid situations when the kernel might *not* reach the initial state. First, some resources can be *cached* either by the TCP implementation or by the Linux kernel. To account for this fact, CMC traces through a few complete TCP connections to generate a state in which the resources are *already* cached. CMC performs the state space exploration from this state. Second, the initial and final states of a TCP connection can still differ due to the various statistics the protocol maintains. During the initial model checking iterations, CMC progressively learns to factor out these variables from the state (after simple manual inspection).

6.3 Checking Implementation Robustness

A TCP implementation should be robust against malformed packets sent by a misbehaving peer. Ideally, we would like to send all possible packets at each implementation state and ensure that the implementation handles them satisfactorily. However, this requires trying a prohibitively large number of transitions (exponential in the size of the packet) at *every* state. Also, most of the packets thus generated will be invalid TCP packets that will be dropped after a few simple checks.

We overcome this problem by generating well-formed TCP packets that are only slightly different from normal packets expected in a particular state. The system consists of two (well-behaved) TCP implementations communicating over an adversarial network. Apart from losing, reordering and duplicating packets, the network can slightly mutate a packet when it is received by the implementation. Mutating a packet includes toggling the control bits in the packet and pruning the packet data to generate very small packets. After performing the mutation, the network ensures that the packet is still well-formed by recomputing the checksum and appropriately modifying the sequence numbers whenever it toggles the SYN and FIN control bits in the packet.

7 Results

The effectiveness of CMC can be measured using two metrics. One is the number of bugs CMC is able to find. The second measure is how well the given system is tested. This section describes our results of model checking the Linux TCP implementation using these two metrics.

7.1 Bugs Found

CMC found four instances where the Linux implementation fails to meet the TCP specification. These errors amply reflect the kind of corner cases CMC is able to test.

The first bug CMC found involves the processing of RST packets. The Linux implementation fails to honor a RST packet in SYN_RCVD state unless the ACK bit is also set. The TCP specification requires that in response to a RST packet an implementation free any resources held by the connection and gracefully inform the application. Figure 1 shows the code containing this bug. The function `tcp_check_req` processes incoming packets in the SYN_RCVD state. The bug was inadvertently introduced while trying to handle a case of a maliciously generated packet. The fix causes the function to prematurely exit before processing the RST flag. This bug can potentially lead to lockup of kernel resources long after the TCP connection is dead.

The second bug involves incorrect handling of a duplicate SYN_ACK packet. While the specification requires that any duplicate packets be ignored, the Linux implementation fails to do so and faithfully uses the acknowledgment to open its congestion window. The error happens because of an incorrect sequence number check in the ESTABLISHED state.

CMC also found that the implementation fails to implement one transition in the TCP state diagram. If an application prematurely closes in SYN_RCVD state, the specification requires the implementation to gracefully close the connection using the *FIN* handshake. An acceptable alternative is to perform an abnormal close by sending a RST packet. CMC found that in this case, the Linux implementation dropped the connection without notifying its peer.

```
// net/ipv4/tcp_minisocks.c
// Process an incoming packet for SYN_RECV
// sockets represented as an open_request.
struct sock *tcp_check_req(...) {
  ...
  /* You would think that SYN crossing is
     impossible here, since we should have
     a SYN_SENT socket (from connect()) on
     our end, but this is not true if the
     crossed SYNs were sent to both ends by
     a malicious third party. We must defend
     against this,
     ...
     Note: This case is both harmless, and
     rare.  Possibility is about the same
     as us discovering intelligent life on
     another plant tomorrow.
     ...
   */
  if (!(flg & TCP_FLAG_ACK))
          return NULL;

  /// BUG: Should have checked the RST field
  ///      before checking the ACK field

  // Invalid ACK: reset will be sent by listening
  // socket
  if (TCP_SKB_CB(skb)->ack_seq != req->snt_isn+1)
          return sk;
  ...
  //  RFC793: "second check the RST bit" and
  //        "fourth, check the SYN bit"
  if(flg & (TCP_FLAG_RST|TCP_FLAG_SYN))
     goto embryonic_reset;
  ...
}
```

Figure 1: The Linux TCP implementation (version 2.4.19) does not honor the RST flag in the SYN_RECV state unless the ACK bit is set. The bug was inadvertently introduced while fixing a “harmless and rare” case. The code prematurely returns when the ACK field is not set on an incoming packet.

	Model Description	Line Coverage	Protocol Coverage	Branching Factor	Additional Bugs
1	Standard server and client	47.4 %	64.7 %	2.91	2
2	Model 1 + simultaneous connect	51.0 %	66.7 %	3.67	0
3	Model 2 + partial close	52.7 %	79.5 %	3.89	2
4	Model 3 + message corruption	50.6 %	84.3 %	7.01	0
	Combined Coverage	55.4 %	92.1 %		

Table 4: Coverage achieved during model refinement. The branching factor is a measure of the state space size.

On examining the source later, we found a comment that acknowledges the incorrect handling of this case.

The final "bug" involves a subtle processing of ACK packets. A TCP implementation is required to abort a connection by sending a RST packet when it receives data that cannot be transferred to the application. This can happen, for instance, when the application closes the connection before the data transfer is complete. The bug involves the case when the implementation receives a packet containing both data and an ACK after the application has closed the connection. The Linux implementation blindly processes the ACK field before processing the data. If the ACK opens the congestion window, the implementation exhibits a peculiar behavior. It sends a stream of data packets and immediately follows with a RST packet aborting the connection. While we are not sure if we should count this as a *bug*, we found this behavior interesting to report.

CMC also detected one instance where the TCP specification might be ambiguous. This concerns the transmission of a FIN packet on a zero window. When the receiver advertises a zero window, the Linux implementation queues a FIN packet, even if no *data* is queued. This is definitely the behavior expected by the TCP specification as the sequence number of FIN lies outside the send window. However, it seems that an acceptable solution is to send the FIN packet as zero window probe.

7.2 Coverage Metrics

While one measure of the effectiveness of a tool like CMC is the number of bugs it finds, another measure is the extent to which a given implementation is tested. We resort to two coverage metrics which are described below. As CMC deals with tremendously large state spaces, CMC typically runs out of resources before completing the search. The coverage metrics are very useful in evaluating the different state space reduction techniques employed by CMC as well as the various models provided by the user.

While various coverage metrics have been studied in software testing [2], we use two metrics that are easy and straightforward to compute. The first measure is the line coverage achieved during model checking. While this measure need not correspond to how well the system has been tested, it is helpful in detecting the parts that are *not* tested.

The second measure, which we call "protocol coverage," corresponds to the behaviors of the protocol tested by the model checker. We calculate protocol coverage as the line coverage achieved in the TCP reference model mentioned above. This roughly represents the degree to which the protocol transitions have been explored.

Table 4 describes the coverage achieved while checking four iteratively refined models. Apart from the two coverage metrics, the table reports the branching factor of the state space as a measure of its (exponential) size. The branching factor is calculated from the number of states seen at a particular depth from the initial state.

The first model described in Table 4 consists of a standard TCP client connecting to a standard server and performing bidirectional data transfer before closing the connection. In the second model, the server nondeterministically decides to initiate the connection, thereby exploring the simultaneous connect mechanism. The third model refines the second model by allowing the client and the server to nondeterministically close the connection before the data transfer is complete. The fourth model introduces an adversary to mutate packets in the network. This tremendously improves coverage at the cost of an increase in state space size. These refinements were made iteratively after investigating the parts of the protocol not covered in a previous model.

The combined coverage in Table 4 reports the coverage achieved cumulatively over the four models. CMC achieves a combined protocol coverage of 92.1%, which represents almost complete coverage of the properties being checked by the TCP reference model. The uncovered lines consist of error condition checks that should not be triggered in a correctly functioning protocol.

8 Related Work

We compare our approach to protocol verification techniques, generic bug finding approaches, and other model checking efforts.

Protocol reliability. Reliability of protocol implementations has been a long running theme in networking research.

One approach is to design a domain-specific language aimed at making networking protocols concise and easy to specify, thereby (hopefully) reducing the chance of error. Examples include ESTEREL [5], LOTOS [4], and Prolac [24]. A drawback of language-based approaches is that, historically, the networking community rarely adopts them — to our knowledge, all widely-used TCP implementations are written in C or C++.

A different but related method aims to reduce errors by providing a more natural infrastructure for networking implementations. Examples include the x-kernel [21], Scout [27] and, to a degree, the Fox project [3]. This approach mainly helps protocol construction, rather than focusing on ways to find errors in the implementation.

There have been numerous attempts to test a TCP implementation. One approach involves transmitting carefully designed packets to the implementation and observing its response [11, 14]. In contrast, x-sim [7] executes an unmodified TCP implementation in a simulator to test for performance related problems.

Complementary to the testing approaches, tcpanaly [33] *passively* analyzes packet traces to detect abnormal behavior of TCP implementations. While this relies on large trace sets to achieve coverage of TCP behavior, this approach has a particularly attractive low up-front cost and scales well to large number of instances.

As stated in the introduction, by using model checking we have a higher up-front cost than these approaches but can explore protocol state spaces much more deeply.

Generic bug finding. There has been much recent work on bug finding, including both better type systems [15, 18, 17] and static analysis tools [13, 1, 9, 29, 16]. While the latter approaches can be easier to apply than model checking (the former can require more manual labor), both are limited to checking relatively shallow rules ("*lock* must be paired with *unlock*"). Model checking can do end-to-end checks out of their reach ("the routing table should not have loops").

Software Model Checking. Several recent verification tools use the idea of executing and checking systems at the implementation level.

Java PathFinder [8] uses model checking to verify concurrent Java programs for deadlock and assertion failures. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program.

SLAM [1] is a tool that converts C code into abstracted skeletons that contain only Boolean types. SLAM then model checks the abstracted program to see if an error state is reachable.

Our goal is to do comprehensive, deep, end-to-end checks of system correctness rather than checking a limited number of properties (as in Pathfinder) or relatively shallow, type-system level ones (as in SLAM).

Verisoft [20] systematically executes and verifies actual code and has been used to successfully check communication protocols written in C. We expect that the techniques we have developed in this paper could be applied to it as well.

As stated in the introduction, we used a prototype version of CMC in prior work [30]. This paper applies it to protocols an order of magnitude more complex, and has led to a complete overhaul of the approach.

9 Conclusions

CMC is successfully able to scale to systems as large and complex as the Linux TCP implementation. Also, CMC has found four bugs in the implementation.

As described in Section 7 and Table 4, CMC is successful in achieving a good coverage of the properties checked. We believe that the results reported in this paper can be improved by using a better reference model that checks for a wider range of properties. For instance, the current model does not check for congestion control properties (that the congestion window should reduce on a packet loss) and timer related properties (e.g. that a retransmission timer is appropriately scheduled for every transmitted packet).

To obviate the need for a separate hand-written reference model, we are currently exploring the possibility of simultaneously executing two different TCP implementations where one can check the behavior of the other. However, there are additional challenges in ignoring acceptable differences in the two implementations.

References

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[2] Boris Beizer. *Software Testing Techniques.* Electrical/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[3] Edoardo Biagioni. A structured TCP in standard ML. In *SIGCOMM*, pages 36–45, 1994.

[4] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. In *Computer Networks and ISDN Systems*, pages 14:25–59, 1986.

[5] Frederic Boussinot and Robert de Simone. The esterel language. Technical report, INRIA Sophia-Antipolis, July 1991.

[6] R. Braden. Requirements for internet hosts - communication layers. RFC 1122, USC/Information Sciences Institute, October 1989.

[7] Lawrence S. Brakmo and Larry L. Peterson. Experiences with network simulation. In *Measurement and Modeling of Computer Systems*, pages 80–90, 1996.

[8] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[9] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[10] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[11] Douglas Comer and John C. Lin. Probing TCP implementations. In *USENIX Summer*, pages 245–255, 1994.

[12] C.Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt, January 2002.

[13] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, 2002.

[14] Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, 1997.

[15] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[16] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[17] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[18] J.S. Foster, T. Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the*

ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, June 2002.

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for single-source shortest path trees. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, pages 112–224, 1994.

[20] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[21] N.C. Hutchinson and L.L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Trans. on Soft. Eng.*, 17(1), January 1991.

[22] Radu Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *Proceedings of 16th IEEE Conference on Automated Software Engineering*, 2001.

[23] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[24] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the prolac protocol language. In *SIGCOMM*, pages 3–13, 1999.

[25] S. McCanne and S. Floyd. UCB/LBNL/VINT network simulator - ns (version 2), April 1999. http://www.isi.edu/nsnam/ns/.

[26] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[27] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.

[28] Madanlal Musuvathi. CMC: A model checker for network protocol implementations. Technical Report PhD Thesis, Stanford University, January 2004. http://verify.stanford.edu/madan/thesis/main.pdf.

[29] Madanlal Musuvathi and Dawson R. Engler. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[30] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[31] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic SPT algorithm based on a ball-and-string model. In *INFOCOM (2)*, pages 973–981, 1999.

[32] G. Nelson. *Techniques for program verification.* Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[33] Vern Paxson. Automated packet trace analysis of TCP implementations. In *SIGCOMM*, pages 167–179, 1997.

[34] Vern Paxson and et.al. Known TCP Implementation Problems. RFC 2525, March 1999.

[35] J. Postel. Transmission control protocol. RFC 793, USC/Information Sciences Institute, September 1981.

[36] U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[37] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[38] The User-mode Linux Kernel. http://user-mode-linux.sourceforge.net/.

Constructing Services with Interposable Virtual Hardware

Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble
University of Washington
`{andrew,rick,mar,gribble}@cs.washington.edu`

Abstract

Virtual machine monitors (VMMs) have enjoyed a resurgence in popularity, since VMMs can help to solve difficult systems problems like migration, fault tolerance, code sandboxing, intrusion detection, and debugging. Recently, several researchers have proposed novel applications of virtual machine technology, such as Internet Suspend/Resume [25, 31] and transparent OS-level rollback and replay [13]. Unfortunately, current VMMs do not export enough functionality to budding developers of such applications, forcing them either to reverse engineer pieces of a black-box VMM, or to reimplement significant portions of a VMM.

In this paper, we present the design, implementation, and evaluation of μDenali, an extensible and programmable virtual machine monitor that has the ability to run modern operating systems. μDenali allows programmers to extend the virtual architecture exposed by the VMM to a virtual machine, in effect giving systems programmers the ability to dynamically assemble a virtual machine out of either default or custom-built virtual hardware elements. μDenali allows programmers to interpose on and modify events at the level of the virtual architecture, enabling them to easily perform tasks such as manipulating disk and network events, or capturing and migrating virtual machine state. In addition to describing and evaluating our extensible virtual machine monitor, we present an application-level API that simplifies writing extensions, and we discuss applications of virtual machines that we have built using this API.

1 Introduction

Virtual machine monitors (VMMs) such as VM/370[8], Disco[6], and VMware [35] have demonstrated that is feasible to implement the hardware interface in software efficiently, making it possible to multiplex several virtual machines (VMs) on a single shared physical machine. Researchers have recognized that VMMs are a powerful platform for introducing new system services, including VM migration [25, 31], intrusion detection [18], trusted computing [17], and replay logging [13]. These services all exploit the unique ability of a VMM to observe and capture all of the events and state of a complete software system, including that of the virtual hardware, the operating system, and applications.

In addition to having this "whole-system" perspective, VMMs have the advantage that the interface they expose is simple in comparison to a operating system API. Virtual hardware events such as disk reads and writes, MMU faults, and network events have a narrow and stable interface, whereas operating systems tend to expose a large number of semantically complex system calls. The virtual hardware interface is an ideal place for deploying many kinds of services: the "whole-system" perspective makes these services powerful, and the simple virtual hardware interface makes them easy to build and able to operate on legacy software.

1.1 The VMM as a Service Platform

A VMM is a compelling platform for deploying an interesting class of system services. Unfortunately, today's VMMs provide precious little support for developing new *virtual machine services.* Because VMMs were not designed to be programmable or extensible, developers have had to reverse-engineer pieces of black-box VMMs to discover or expose the interfaces they need [25], or in extreme cases, they have had to reimplement significant portions of the VMM [13]. Worse, because there is no standard interface or extension framework supported by today's VMMs, VM service authors have not been able to cooperate with each other or re-use developed components. For example, ReVirt's replay ability [13] and Hypervisor-based fault tolerance [5] are both based on a similar logging primitive, but it would be difficult for these two projects to share this common functionality.

The current state of virtual machine services is similar to the distributed computing era before the advent of standardized transport protocols and remote procedure calls [4]. In this era, programmers used a variety of ad-hoc, home-grown techniques and services to communicate across machine boundaries, resulting in brittle, unreliable, and non-interoperable systems. Standardized transport and RPC allowed distributed systems programmers to focus on the logic of their systems, relying on underlying plumbing to solve common issues of reliable communications, naming, and type marshalling.

In this paper, we attempt to advance the state of VM service construction by exploring two questions. First, what should the programmatic interface exposed by VMMs to VM services look like? Second, what structure and mechanisms within a VMM are needed to support this interface well?

To address the first question, we propose a high-level software toolkit that allows programmers to build services in one VM that **interpose** on the events generated by another VM's virtual hardware devices, or to **extend** the virtual hardware exposed to virtual machines by implementing new virtual hardware devices. These two abilities (interposition and extension) allow programmers to develop software services that manipulate the virtual machine interface without worrying about the underlying virtualization mechanisms and plumbing. Because our toolkit exposes a well-defined API, extensions and interposition services would work on any VMM that exposes the same API.

To address the second question, we describe the design and implementation of a virtual machine monitor that supports virtual device interposition and extension. Our VMM, which we call μDenali, is built around a flexible virtual hardware event routing framework that borrows significant pieces of its design from Mach ports [12]. In our framework, virtual hardware events are defined as typed messages, and virtual hardware devices are simply sets of ports. Interposition is achieved by re-routing messages from one virtual machine to another. Extensibility is achieved by allowing a VM to expose ports that send and receive appropriately typed messages.

The remainder of this paper is structured as follows. In Section 2, we briefly describe the Denali VMM [36], which we modified to build μDenali. Section 3 gives an architectural overview of μDenali, describing the basic structure of the system and a high-level view of the extensibility and interposition interface which it exposes. In Section 4, we focus on the port-based routing framework within μDenali, and we describe the virtual hardware underlying VMs in terms of the message types and ports that define each virtual device. In Section 5, we describe a number of virtual hardware device extensions and virtual machine services which we have implemented. We present an evaluation of μDenali in Section 6, we discuss related work in Section 7, and we conclude in Section 8.

2 Denali Overview

The Denali isolation kernel [36] is an x86-based virtual machine monitor whose goal is to support a large number of concurrent virtual machines. Denali is a type-I VMM, meaning that it runs directly on physical hardware, as opposed to type-II VMMs (such as VMware workstation [35] or user-mode Linux [11]) which run as applications on a host operating system.

Denali relies on a technique called para-virtualization to enhance its performance, scalability, and simplicity. Rather than exposing a virtual architecture that faithfully reproduces the underlying physical architecture, Denali strategically modifies the virtual architecture, retaining the performance advantages of direct instruction execution but modifying key features of the virtual architecture such as interrupt processing, handling non-virtualizable instructions, and timers.

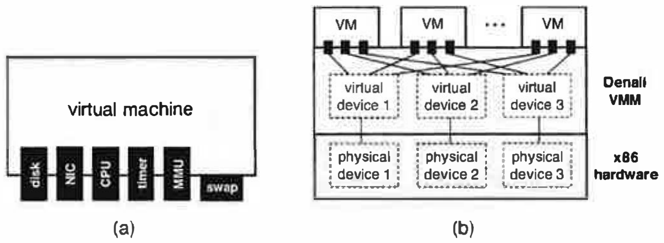


Figure 1: **Denali virtual machine monitor architecture.** The "old" pre-extensible architecture. Each virtual machine interacts with the virtual machine monitor through virtual hardware devices; each hardware device is hard-wired to a virtual device implementation inside the Denali kernel.

The Denali implementation has progressed significantly since what was reported in [36]. Most notably, because of the addition of a virtual MMU device, Denali is now able to run full-fledged operating systems in a manner similar to Xen [2]. Most modern OSs run on multiple architectures, isolating the architecture-dependent pieces of the implementation from the architecture-independent pieces. We successfully ported the NetBSD operating system to Denali by implementing architecture-dependent components appropriate for the Denali virtual architecture. Because of differences between our virtual MMU and the x86 MMU, we cannot run x86 binaries directly, but with recompilation we can run NetBSD and any of its applications.

2.1 Denali's Architecture

Figure 1 illustrates the Denali virtual architecture. The interface between each virtual machine and Denali is a set of virtual hardware devices. Within the Denali VMM, these virtual devices are "hard-wired" to virtual device implementations that (1) multiplex the virtual devices of the many VMs onto their physical counterpart (including namespace virtualization) and (2) implement physical resource management policies. Denali supports the following virtual devices:

Virtual CPU: The virtual CPU executes the instruction streams of virtual machines, emulates privileged instructions, exposes virtual interrupts, and handles virtual programmed I/Os. The virtual CPU also exposes a number of purely virtual registers, such as a register which contains the current wall-clock physical time.

Virtual MMU: The Denali virtual MMU exposes a software-loaded TLB with a fixed number of virtual address-space IDs (ASIDs). The virtual MMU is a completely different abstraction than the underlying x86 hardware-based page tables; we made this design choice to simplify the virtualization of virtual memory. Borrowing terminology from Disco [6], we refer to true physical memory as *machine memory*, virtualized physical memory as *physical memory*, and virtualized virtual memory as *virtual memory*. The virtual MMU allows guest OSs to implement multiple virtual address spaces, and to page or swap between virtual memory and physical memory. Denali itself overcommits physical memory and implements swapping between machine and physical memory transparently to VMs.

Virtual timers: Denali exposes a virtual timer that supports an "idle-with-timeout" instruction, allowing virtual machines to relinquish their virtual CPUs for a bounded amount of time.

Virtual network: The Denali virtual NIC appears to guest OSs as a simple Ethernet device. The NIC supports packet transmission and reception at the link level.

Virtual disk: Stable storage is exposed to guest OSs as a block-level virtual disk. Guests can query the size of the disk, and read and write blocks to disk.

Because it supports neither extensibility nor interposition, implementing the VM services that have appeared in recent literature would be difficult using Denali. For example, to checkpoint and migrate a virtual machine, all virtual device state must be captured, including virtual registers, physical memory pages (whether resident in machine memory or swapped to physical disk by Denali), virtual MMU entries, virtual disk contents, and any virtual disk operations that are in flight. However, none of this state is exposed outside of the VMM. In the next section of this paper, we describe the architecture of μDenali, a significant reimplementation of the Denali VMM which supports both extensibility and interposition to support these kinds of services.

3 μDenali: an Extensible VMM

A VMM provides three basic functions: it virtualizes the namespace of each hardware device in the physical architecture, it traps and responds to virtual hardware events to implement virtual hardware device abstractions, and it manages the allocation of physical resources to virtual machines. In the Denali VMM, these three functions were co-mingled. The major insight of μDenali is that these functions can be separated from each other, and by doing so, virtual hardware event interposition and extensibility become a simple matter of routing one VM's virtual hardware events to another VM.

Physical resource management: μDenali borrows significant code from Denali to interact with and manage physical devices. We utilize low-level code and device drivers provided by the Flux OSKit [15], and we implement our own global resource management policies across VMMs, such as round-robin CPU scheduling across VMMs, fair queuing of received and transmitted Ethernet packets, and a static allocation of physical disk blocks. These global resource management policies, which we do not allow to be extended or overridden, provide performance isolation between VMs.

Device namespace virtualization: Each virtual device in the Denali architecture is capable of generating and receiving several device-specific events. For example, virtual disks receive disk read and write requests, and generate request completion events. As another example, virtual CPUs generate interrupts and faults. In μDenali, we associate a typed message with each of these device-specific events. Implicit in the message type definitions are the namespaces associated with devices: disk event messages contain block offsets, and CPU faults contain the memory address of the faulting instruction.

Virtual hardware event trapping and routing: Denali directly executes the instructions of virtual machines, but privileged instructions and virtual hardware device interactions are trapped to and emulated by the VMM. In μDenali, we borrow Denali code to trap on these events, but instead of directly handing them off to virtual device implementations within the VMM, we encode these events within typed messages and route the messages to an endpoint. μDenali contains a routing infrastructure which associates destination **ports** with events generated by the hardware devices of each VM. Default virtual hardware devices within μDenali have ports, as do special **interposition devices** associated with each VM. By binding a hardware device of one VM (the child) to the interposition device of another (the parent), the parent VM gains the ability to interpose on the child's virtual device. The parent VM can either reroute the events back to μDenali's default virtual device implementation, or handle them itself, in which case the parent becomes an extension to the virtual architecture of the child.

Figure 2 illustrates the μDenali architecture. In Figure 2(a), we show the structure of a virtual machine. Each VM sees a collection of virtual hardware devices with which it interacts, similar to Denali. Unlike Denali, a μDenali VM can also interact with its interposition device. A parent VM receives virtual hardware events from children VMs on which it interposes, and the parent can send response events (either to μDenali or to the child) over this device. Additionally, the interposition device exposes various control functions to a parent, such as the ability to instantiate a new child VM, suspend or

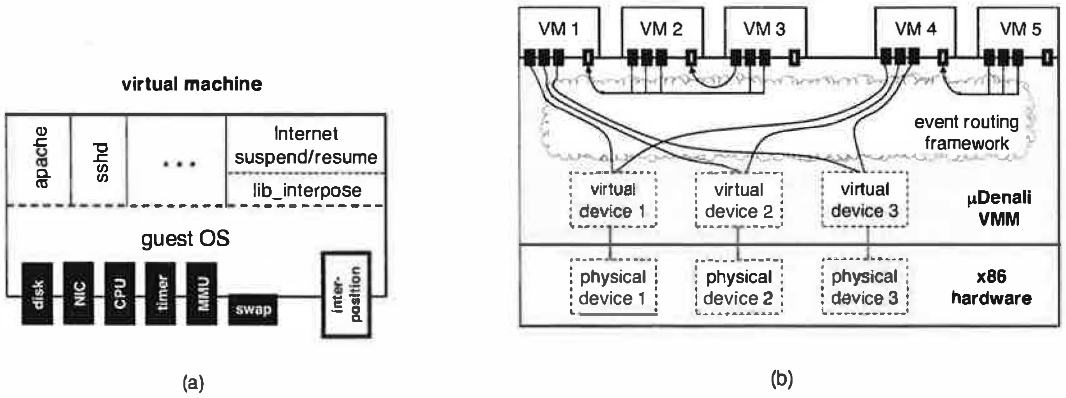


Figure 2: **μDenali virtual machine monitor architecture.** (a) A new virtual hardware device, the "interposition device", is visible to each VM. The guest OS exposes events routed through this interposition device to user-level virtual machine services, like Internet suspend/resume, with the help of a C library called *lib_interpose*. (b) A VM's virtual hardware devices can be bound to other VMs' interposition devices, or to default virtual devices implemented in the VMM. An event routing framework (based on Mach ports [12]) routes virtual device events to appropriate endpoints, enabling both event interposition and virtual device extensions.

resume execution of a running child VM, or extract the hardware state of a child VM from μDenali.

The guest operating system within a virtual machine can choose to handle virtual hardware and interposition device events as it sees fit. In our port of the NetBSD operating system to μDenali, we have written NetBSD device drivers for the "regular" virtual hardware devices, and we have exposed the interposition device to user-level applications through a high-level interposition library written in the C programming language. Using this interposition library (*lib_interpose*), user-level applications can implement virtual machine services, such as Internet Suspend/Resume, which manipulate the device state and events generated by other VMs.

In Figure 2(b), we show the μDenali VMM architecture. Similar to Denali, μDenali runs directly on x86 hardware, and contains a set of default virtual device implementations. μDenali also contains an event routing framework, which manages bindings between ports associated with virtual hardware devices of running VMs, and routes typed messages according to these bindings. The figure shows five virtual machines. VM1 is the parent VM of two children (VM2 and VM3). VM1 receives and handles all events generated by all virtual devices of VM2, and some virtual devices of VM3. VM2 is a partial parent of VM3, receiving events from the remaining virtual devices of VM3. VM1 has no parent, meaning that all of its virtual device events are bound to default virtual devices within μDenali. Similarly, VM4 is the parent of VM5.

3.1 The NetBSD Interposition Library

The μDenali interposition device defines the set of extensibility and interposition operations that a parent VM can perform on a child. The interface to this device consists of a set of downcalls that a parent initiates when it wishes to perform an action on a child (for example, to instantiate a new child VM), and a set of events and responses that are exchanged between the parent and μDenali when an event of interest happens within the child VM.

The interposition device interface is implemented in terms of architecture-level programmed I/O operations and interrupts. Our NetBSD interposition library, with the help of the NetBSD guest OS, exposes this interface to applications in terms of function calls in the C programming language. In the remainder of this section, we describe the key components of the interposition device interface by showing fragments of the interposition library API.

One key design decision we made was to focus on *local* extensibility and interposition—that is, the interposition device only exposes events generated by child virtual machines. We do not expose global events, such as μDenali scheduler decisions or machine memory allocation and deallocation events. Because of this, the μDenali extensibility framework does not suffer from security issues plaguing extensible operating systems, which permit extensions to modify global system behavior [3]. μDenali extensions are isolated from the μDenali VMM and from other VMs as a side-effect of being implemented inside their own virtual machine, and extensions can only affect children VMs.

3.1.1 Virtual Machine Control

The control API allows a parent VM to create, destroy, start, and stop child virtual machines. Figure 3 shows a subset of the NetBSD C library functions and

```
typedef struct {

// the swapDevice provides backing store for the
// child's physical pages when Denali swaps.
SwapDevice *swapDevice;

// a Disk provides block-addressable storage.
Disk *disk;

// (other virtual devices omitted, for brevity.)
} VirtualMachine;

// create and destroy child VMs.
int createVM(VirtualMachine *vm, char *macAddr);
int destroyVM(VirtualMachine *vm, char *macAddr);

// associate a local /dev/virtethX block device
// with a new virtual Ethernet in the child.
int createEthernet(VirtualMachine *vm);

// suspend and resume a child VM.
int suspendVM(VirtualMachine *vm);
int resumeVM(VirtualMachine *vm);
```

Figure 3: **Virtual machine control functions.** The VM control portion of the interposition API allows parent VMs to create, destroy, start, and stop child VMs.

structures associated with these control operations.

The create and destroy operations permit a parent to dynamically assemble a child out of constituent virtual hardware devices, and then cause μDenali to instantiate and begin executing the assembly. For each device type, the parent must either provide callback functions to handle messages associated with that device, or provide an alternate routing port (such as a parent VM's port, or μDenali default device ports) for the device to bind to.

The suspend and resume operations act on already active virtual machines. These commands are often used in conjunction with other interposition commands. For example, a checkpoint application would stop a virtual machine before extracting the checkpoint to ensure that it obtains a consistent snapshot of virtual device state.

3.1.2 I/O Device Interposition

The I/O device API allows a parent VM to interpose on virtual device events raised by child VMs, and to either re-route or respond to those events. This API allows parents to interact with virtual disks, Ethernets, and swap store backing (virtual) physical memory. As in Denali, μDenali device events are simplified idealizations of the underlying physical devices. For example, the virtual disk interposition interface supports two principle operations: reading and writing disk blocks. Figure 4 shows a representative subset of the I/O device API.

We chose to special-case the interposition interface

```
// the virtual Disk device callback functions
typedef struct {

  // the child generated a write event.
  int (*diskWrite)(char *buffer, int offset,
                   int num_sectors);

  // the child generated a read event.  If the
  // parent chooses to handle the event, it
  // puts the appropriate data in "buffer".
  int (*diskRead)(char *buffer, int offset,
                  int num_sectors);

  // the child is asking the disk to report
  // how many sectors it contains.
  int (*getSectorSize)(void);

} Disk;
```

Figure 4: **I/O device interposition functions.** The I/O device interposition API permits parents to interpose on and respond to their children's device operations. In this figure we show only the interface associated with the virtual disk: other devices have similar interfaces.

to children's virtual Ethernet devices. The interposed Ethernet is exposed to the parent as a network device within NetBSD, (e.g., `/dev/virteth0`). This allows us to reuse existing networking software like NAT proxies, routers, and intrusion detection systems within a virtual machine service. From the point of view of the parent VM, it is connected to its child over a a dedicated network interface.

3.1.3 Exposing μDenali Internal State

In practice, not all virtual machine state can be directly exported through the interposition device. For performance reasons, we chose to cache run-time state such as hardware registers, MMU mappings, in-flight virtual device operations, and the status of swap buffers inside the μDenali kernel itself. However, to implement services such as checkpoint and migration, parent virtual machines must be able to flush and access this state upon request. In practice, only "hard" state must be saved. For example, Ethernet packets that are queued for transmission or delivery inside μDenali can be discarded, because doing so is consistent with the best-effort semantics of Ethernet networks.

Because much of this run-time state is platform specific, μDenali encapsulates it inside in an opaque data structure. Extensions such as debuggers that require low-level access to platform-specific structures must manipulate this run-time state with knowledge of μDenali's encapsulation syntax. Figure 5 shows a subset of the flush and state-capture functions that are exposed by the interposition device.

```
// functions to flush cached state from
// Denali, and extract/restore kernel
// internal associated with a VM.

int flushSwap(VirtualMachine *vm);

int extractMMUmappings(VirtualMachine *vm,
                       MMUState *ms);
int restoreMMUmappings(VirtualMachine *vm,
                       MMUState *ms);

int extractCPUstate(VirtualMachine *vm,
                    CPUState *cs);
int restoreCPUstate(VirtualMachine *vm,
                    CPUState *cs);
```

Figure 5: **Flushing and extracting VMM state.** μDenali internal state associated with a VM can be flushed and extracted using this part of the API. Here, we only show a portion of the full API.

3.1.4 Tracking Non-Determinism

Tracking and logging non-deterministic events is necessary to implement replay services such as ReVirt [13]. We have not yet finished the implementation of this functionality, but we have a complete design that takes advantage of μDenali's message routing layer.

To facilitate logging and replay services, μDenali must expose two types of information: the precise instruction-level timing of asynchronous events (such as virtual interrupts), and the content of events that are obtained from external input, such as user keystrokes or network packets. Both types of information are readily available in the μDenali architecture. Asynchronous events are defined to occur when the message routing infrastructure delivers an event to a VM, and the content of external events can be observed through interposition. For example, an external Ethernet packet arrives when a virtual Ethernet packet message is delivered to a virtual Ethernet device, and the content of that message can be observed and logged by a parent through interposition.

In our design, μDenali assists with the logging of non-deterministic events by recording the timing of messages delivered to a child VM, and periodically flushing this log to the parent's interposition interface. Each log entry contains a three-tuple: a unique event ID, the event type (which is simply the type of the message delivered to the child port), and a delivery time. As with ReVirt, delivery time is measured by a software instruction counter [27] which includes the instruction address and a counter of the number of backward branches executed during the lifetime of the child VM.

Replay requires the ability to interrupt a child VM before the execution of a logged non-deterministic event. To support this, the parent VM must be able to pass a previously recorded log file to μDenali, which uses the log to trap the child VM at the precise moment that a logged non-deterministic event should occur. Our implementation of logging and replay is underway, but not complete.

3.2 Summary

In this section, we described the architecture of the μDenali extensible virtual machine monitor, and presented the high-level interposition and extension interface that it exposes. This interface, which can be accessed through an application-level library implemented in the C language, provides programmers with a powerful and natural way to implement new virtual machine services. In the following section of the paper, we drill down into the design of the message routing infrastructure inside μDenali.

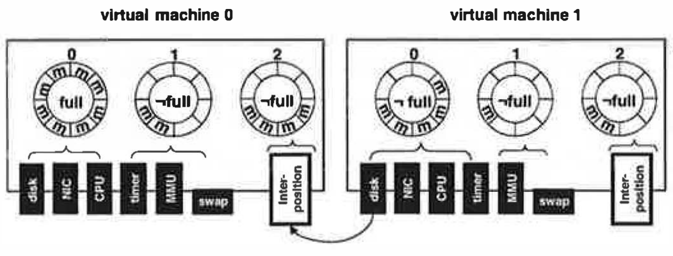
4 Event Routing in μDenali

As described in the previous section, μDenali contains an event routing framework that is responsible for receiving, routing, and delivering virtual device events. In essence, the event routing framework is a virtual bus that provides a communication channel between a VM's virtual devices and the implementation of those devices.

Our routing framework design was inspired by Mach ports and messages [12]. A port is an abstraction of a protected communication channel that facilitates the reception and transmission of typed messages and port capabilities. Each virtual device in each virtual machine has a set of ports associated with it, and ports correspond to operations supported by virtual devices. By binding a virtual device port from a child VM to the interposition device port of a parent VM, events generated by that child's virtual device are converted into typed messages and delivered to the parent. Figure 6 illustrates the routing framework design.

4.1 Port Capabilities and the Port Table

Ports are protected by capabilities. Like Mach, μDenali defines three types of capabilities: the receive right allows the holder to receive messages delivered to that port, the send right allows the holder to send messages to the port, and the send-once right allows the holder to send a single message to the port (for example, to allow a message recipient to send a reply). A capability can be transmitted over ports in order to grant a privilege to the recipient. Only send rights can be copied; if a send-once right is passed in a message, the sender loses that right. The holder of a receive right (defined to be the port owner) may create send and send-once capabilities for that port.



(a)

VM 1's port table

port name	port rights	port type	refcount	ringbuf
0	R	disk read reply	0	VM1:0
1	R	disk write reply	1	VM1:0
2	S	disk read request	1	VM0:2
3	S	disk write request	1	VM0:2

(b)

Figure 6: **μDenali port tables.** (a) VM1's virtual disk is managed by VM0. (b) A subset of VM1's port table. Because VM1 can generate virtual disk read and write requests, VM1 has send rights on the ports associated with those operations (ports 2 and 3). VM0 manages the associated receive ports, and uses its ring buffer #2 to buffer messages arriving on it. Because VM0 manages the disk, it needs to be able to send disk read and write replies back to VM1. VM1 has receive ports (ports 0 and 1) to receive these replies, and it uses its ring buffer #0 to buffer disk reply messages.

The μDenali VMM maintains a table of port capabilities on behalf of each virtual machine. Because this port table is managed inside the VMM, port capabilities are unforgeable, similar to UNIX file descriptors. **From the point of view of μDenali, a virtual machine is simply a port table.** Much like a MCS capability list [10] or Hydra local name spaces [37], a port table simultaneously defines the operations that can be performed on an object (virtual devices) and the namespace in which those objects are embedded (the virtualized namespaces of virtual devices).

When a parent VM creates a new child VM, the parent sets up the initial port table of the child. To do this, the parent creates individual ports in the child's port table, grants port capabilities as necessary, and binds the ports as appropriate. A parent can bind a child's port to its own interposition device, to the interposition device of another VM for which it has a port send capability, or to the port associated with the default virtual device implementations inside μDenali.

4.2 Port Queues and Message Buffers

Like hardware buses, the μDenali event routing framework itself never stores messages. All message queuing must be implemented by the virtual device which owns the port which receives the message. To accomplish this, each virtual machine maintains one or more ring buffers associated with its ports. When a message arrives on a port, μDenali atomically delivers and advances the ring buffer. If the ring buffer is full, all ports associated with it enter the "full" state, and error messages are returned to source ports on future message delivery attempts. When the ring buffer is drained by the virtual machine, the receive ports associated with it leave the full state, and notifications are sent to all send ports bound to them.

Because the event routing framework has no storage, providing atomic message delivery is simple: the routing framework promises either successful delivery, or an immediate error. This lack of storage also means that the routing framework itself is stateless, meaning that it need not be involved in the checkpoint or recovery of a virtual machine. Note, however, that port tables and ring buffers must be included in checkpoints.

μDenali messages contain a small amount of in-band data (16 32-byte words). For several of our devices, such as virtual MMUs, this suffices. For those that deal with larger blocks of data (e.g., virtual disks), μDenali provides an out-of-band data passing mechanism to transfer up to 64KB of data with each message. This out-of-band channel is similar in spirit and usage to direct memory access (DMA) on modern physical hardware devices.

4.3 Example Virtual Devices, Ports, and Message Types

μDenali defines a standard set of virtual devices and associated operations and events. Operations and events are exposed to virtual machines by the routing framework by ports and message types, respectively. We now discuss a few virtual devices and their ports and message types. Note that there is a one-to-one correspondence between ports and messages in μDenali's event routing framework, and function calls and arguments in the *lib_interpose* interposition library described in Section 3.1. Though we only discuss a subset of the virtual devices supported by μDenali, the other virtual devices have similar and consistent designs.

4.3.1 Virtual CPU

A VM's virtual CPU device (vCPU) is responsible for executing the instructions of the guest operating system and its applications, managing the registers of VMs, and exposing ports and messages to VMs. All virtual machines currently rely on the default vCPU implementation inside μDenali.

The vCPU shares most of its specification with the underlying x86 physical CPU. With the exception of a handful of non-virtualizable instructions [29] which have undefined (though secure) results, all unprotected instructions are available to the guest. All of the unprotected x86 registers are also available to the guest.

μDenali augments the x86 CPU with a set of purely virtual instructions and registers that have no physical counterpart, as discussed below.

The vCPU maintains each VM's port table, and also handles the transmission and reception of messages on ports. To transmit a message, a VM places the message and destination port-descriptor in a fixed memory location and invokes a virtual instruction. The vCPU device traps this instruction, type-checks the message against the port table, swizzles any port references in the message, converts the physical addresses referring to out-of-band data into machine addresses, and attempts to place the message in the receive port ring buffer. From the point of view of the VM transmitting the message, this process corresponds to a single atomic instruction (albeit a high latency one). On reception of a message, the vCPU interrupts the receiver VM and jumps to a guest-specified interrupt handler, assuming virtual interrupts are enabled.

The vCPU provides the mandatory implementation of all VMs' virtual MMU devices. We implemented this device in μDenali because it is the most frequently used device in the system, and its implementation is closely intertwined with physical page table and machine memory management. We have not yet considered whether it is possible or pragmatic to permit virtual MMUs to be interposed on or extended by parent VMs.

4.3.2 Virtual Swap Device

Interposing on the virtual swap device permits a parent VM to manage the swap file backing a child VM's physical memory pages. μDenali decides when to reclaim a machine page that is bound to a physical page of a VM, and when it does so, it generates swap events on behalf of the child. A swap device is conceptually similar to external pagers in Mach and V [38, 7]. By interposing on a swap device, a parent has the ability to checkpoint all of the physical memory of its child.

A swap device has three ports, and supports four message types. Whenever a VM generates a physical-to-machine page fault within μDenali, the VMM generates a *fault* message and sends it to the VM's fault receive port. The handler is responsible for processing the message and sending a reply back when the fault has been serviced. When μDenali needs to swap out a physical page, it generates a *swap-out* message and sends it to the swap receive port of the VM. Finally, a parent VM can generate a *flush* or *flush-all* message to flush one or all dirty physical pages back to the swap device.

4.3.3 Virtual Ethernet Device

The virtual Ethernet device manages the transmission and reception of Ethernet packets. A virtual Ethernet device has two ports, and supports one message type. When an Ethernet frame arrives for an Ethernet device, an *Ethernet packet* message is created and sent to the virtual Ethernet device's receive port. When a VM transmits an Ethernet frame, another *Ethernet packet* message is created and sent on the virtual Ethernet device's send port.

A VM's virtual CPU facilitates the creation of new virtual Ethernet devices. A parent VM can send a *create new virtual Ethernet* message to the Ethernet creation port of the virtual CPU, causing a new virtual Ethernet device to be created with a specific MAC address.

4.4 Summary

In this section, we described μDenali's event routing framework, whose design borrows heavily from Mach ports and messages. We also described the ports and message types associated with virtual devices supported by μDenali. In the next section of the paper, we describe several virtual device extensions and virtual machine services that we have implemented on μDenali.

5 Application Case Studies

In this section of the paper, we demonstrate how μDenali's extensibility and interposition mechanisms can be used to develop powerful virtual device extensions and virtual machine services.

5.1 Internet Suspend/Resume

The Internet Suspend/Resume project [25] uses the Coda file system to migrate checkpointed VMware virtual machines across a network. We have re-implemented this functionality in μDenali, albeit using the less efficient NFS rather than Coda.

Migrating a VM consists of three phases: checkpointing and gathering the complete state of the VM, transmitting this state over the network, and unpackaging the state into a new VM on the remote host. In μDenali, we use virtual device extensions to maintain and gather the state of a child VM's virtual devices (for example, a swap disk extension maintains the complete physical memory image of the child), and we rely on the state extraction routines in the interposition library to extract μDenali internal hard state associated with the child.

To transmit VM state over a network, we rely on the NFS file system. The parent stores all gathered state inside an NFS file system also accessible to the remote host on which the VM will be resumed.

We found the development of migration on top of `lib_interpose` to be straightforward. The entire implementation consists of 289 C source lines. The bulk of this complexity is dedicated to serializing C structures into a byte stream, a task handled automatically by more modern languages such as Java.

5.2 "Drop-in" Network Services

As an example of a virtual machine service that exploits network device interposition, we have created a dynamically-insertable network intrusion detection system (IDS) embedded within a virtual machine. Our system (which we call VM-snort) acts as a parent that interposes on other VMs' virtual Ethernet devices. Any network packet flowing in or out of a child VM must pass through VM-snort. We developed VM-snort using the Snort network intrusion detection system.

Because VM-snort is embedded in a virtual machine checkpoint, it is possible for a third party to create a "pre-packaged" IDS image that can be dynamically instantiated at customer sites, using the checkpoint/restore functionality we previously described. Thus, VM-snort is the virtual equivalent of a dedicated physical intrusion detection appliance.

This model of embedding a network service inside a VM is generalizable to other network services. Many active services (i.e., those that modify in addition to monitor packets) are possible, such as drop-in VPN servers, transparent Web proxy caches, or application-level firewalls. Other passive services besides VM-snort are also possible (i.e., services that only passively monitor packets), such as a dynamically deployed distributed worm detector or an Internet weather service sensor.

5.3 Continuous Rejuvenation

In practice, it is extremely difficult to remove all bugs from complex software systems. Many bugs in production systems are "heisenbugs" [20], which depend on intricate sequences of low-probability events. Other errors like memory leaks can emerge slowly, thereby defeating many testing procedures. Researchers have proposed pro-active restarts to forestall the effect of these latent errors [22]. However, simply restarting a machine creates short-term unavailability, which may be unacceptable for some services. Another alternative is to replicate the service across physical machines, but this can increase cost and administrative overhead.

By using restartable *virtual machines*, we can achieve many of the benefits of software rejuvenation without the accompanying downtime. To demonstrate this, we have constructed a service called *Apache**, which serves web requests from a (virtual) web server farm. The system consists of K child nodes, each running a standard copy of the Apache web server. The parent VM acts as a layer-3 switch, redirecting incoming requests to the child that is active at any given time. After a configurable time period, the parent VM redirects requests to the next child in the pipeline, and reboots the child[1].

*Apache** leverages μDenali's interfaces for disk, swap, and network interposition. Our implementation was greatly simplified by leveraging NetBSD's built-in NAT functionality to handle request distribution. We use an implementation of copy-on-write disks (described below) to isolate state changes inside the Apache virtual machines. The entire implementation (including COW disks) required 1131 lines of C code.

5.4 Disk and Swap Device Extensions

Because μDenali virtual swap devices are extensible, implementing novel bootloaders is simple. As a demonstration of this, we implemented a network boot loader, which fetches the child's swap image over HTTP. The boot loader saves the swap image in the NetBSD file system of the parent. When swap-in or swap-out events are generated by the child's virtual swap device, the bootloader serves the appropriate frames from this file. Our bootloader makes use of the `wget` Unix program to fetch files from Web servers. We implemented the portion of the bootloader that handles swap events and serves frames from a file; this required 94 lines of C code.

Using μDenali's disk interposition mechanism, we built a simple copy-on-write (COW) disk extension. Our COW disk extension permits multiple VMs to share a base disk image, but exposes the abstraction of a private mutable disk per VM. Like all standard COW implementations, our extension maintains a file containing differences between the base disk image and each VM's current disk image. A COW disk is useful in many situations, including efficiently creating multiple "cloned" virtual machines on a single physical machine. Our copy-on-write disk extension consists of 675 lines of C code.

We are currently working on a "time-travel" disk implementation, which records all disk updates to a log. This provides a recovery mechanism that allows a system to roll back to a previous working state.

6 Evaluation

Our evaluation explores three aspects of μDenali's performance. First, we measure the basic overhead introduced by μDenali's virtualization and extensibility. Next, we measure the performance of interposed virtual hardware devices (such as interposed disks and Ethernets), and compare it to non-interposed virtual devices. Finally, we measure the performance of our Internet Suspend/Resume and VM-snort services.

For our experiments, μDenali ran on a 3.2 GHz Pentium 4 with 1.5 GB of RAM, an Intel PRO/1000 PCI

[1]Actually, there is a delay before rebooting a virtual machine to allow existing connections to terminate.

operation	latency
native NetBSD null system call	0.38 μsec
μDenali NetBSD null system call	1.2 μsec
virtualization overhead of μDenali's virtual Ethernet (send, without interposition)	2.0 μsec
virtualization overhead of μDenali's virtual Ethernet (send, with interposition)	7.2 μsec

Figure 7: **VMM overhead.** This table compares microbenchmarks run on Linux executing directly on physical hardware, and on our ported NetBSD executing on μDenali. The "virtualization overhead of μDenali's virtual Ethernet" latencies show the overhead introduced when packets are sent through a virtual Ethernet device, excluding the physical Ethernet send costs. The cost of the interposition machinery is shown by comparing the "without interposition" and "with interposition" cases.

gigabit Ethernet card connected to an Intel 470T Ethernet switch, and an 80 GB IDE hard drive running at 7200 RPM. For any experiment involving the network, we used a 1500 byte MTU. We used the httperf [28] Web workload generation tool.

Our Ethernet card is not well-supported by conventional NetBSD on physical hardware. Therefore we compare μDenali's NetBSD performance with Linux in the results that follow.[2] We do not believe this qualitatively changes our results.

6.1 Basic Overhead

A major source of overhead for VMMs is the trapping and emulation of privileged instructions. For μDenali, these instructions include virtual hardware device programmed I/O instructions and system calls issued by a guest OS. The event routing framework in μDenali imposes additional overhead on interposed virtual device operations.

Figure 7 shows the cost of commonly used operations. Null system calls (getpid) are more than three times as expensive on μDenali as on conventional NetBSD. In addition to trapping on the original system call, μDenali must trap on the system call return. Our trap handling code is based on old Mach code, and has not been as carefully optimized as NetBSD.

Network communication is more expensive on a VMM because packet send operations must be trapped and emulated. μDenali incurs an overhead of 2.0 microseconds to send a 1400 byte packet through a (non-interposed) virtual Ethernet, relative to a physical Ethernet driver. This overhead includes the cost of a kernel trap and packet copy. For interposed Ethernets, the

[2]Note that NetBSD running on μDenali can exploit our physical gigabit card, since our VMM interacts with the true physical device and has the correct drivers, while a NetBSD VM interacts with a simple virtual device.

	Apache, 2KB	Apache, 64KB	Apache, 128KB	Bulk TCP
Linux	6700 req/sec	2300 req/sec	614 req/sec	728 Mb/sec
μDenali NetBSD, no interposition	3200 req/sec	1400 req/sec	550 req/sec	684 Mb/sec
μDenali NetBSD, null interposed Ethernet	2200 req/sec	880 req/sec	325 req/sec	387 Mb/sec
μDenali NetBSD, Snort interposed Ethernet	1350 req/sec	344 req/sec	116 req/sec	214 Mb/sec

Figure 8: **μDenali network overhead.** Moving from left to right, these tests transition from system-call intensive to packet-intensive. μDenali performs better in the packet-intensive regime. The 3x system call penalty we previously described contributes to a substantial performance loss for small transfers, and a moderate performance loss for large transfers.

message routing framework within μDenali introduces an additional cost of 5.2 microseconds, which includes the cost of a message transfer, a context switch to the "parent" VMM, and message reception. This value represents an upper bound because μDenali is capable of batching multiple messages per receive operation.

6.2 Device Interposition Overhead

In this section, we evaluate the performance of μDenali's virtual network and disk devices, and compare the performance of these devices with and without interposition. To evaluate network performance, we benchmarked the Apache Web server serving static documents of various sizes. We also measured bulk TCP throughput. Figure 8 shows results for four configurations: 1) Native Linux, 2) μDenali NetBSD without Ethernet interposition, 3) μDenali with "null" Ethernet interposition, which provides simple packet forwarding via a parent VM, and 4) μDenali with Snort running inside the interposing VM. We use the default Snort rule set. Our generated traffic stream consisted of normal (non-malicious) Web requests, so Snort performed very little logging.

Figure 9 shows the same network performance results graphically by normalizing μDenali's performance against Linux. Without interposition, μDenali's performance degrades relative to Linux as the TCP transfer size decreases. μDenali obtains 94% of Linux's bulk TCP bandwidth, but only 48% of Linux's 2k Web throughput. This discrepancy arises because TCP connection setup is more expensive on μDenali, owing to the 3X system call performance gap. For bulk TCP throughput and large Web documents, there are fewer system calls, and μDenali performs competitively with Linux.

Although the virtualization overhead is highest for short TCP transfers requests, the additional overhead

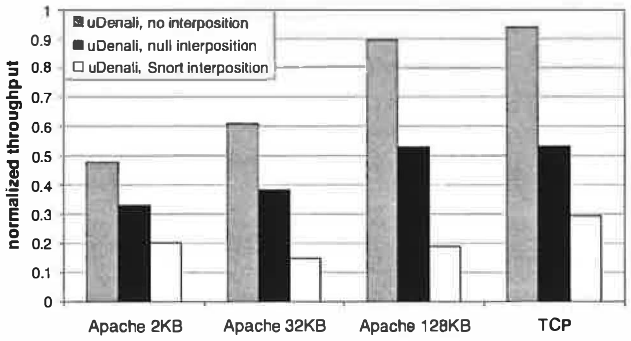


Figure 9: **Network interposition overhead:** Results are normalized against a native Linux machine. For small transfers, system call costs dominate. For large transfers, per-packet costs dominate. Interposing on network devices adds to per-packet latencies, resulting in a relative performance drop for large transfers.

	Sequential Read Throughput
Linux	45 MB/sec
μDenali NetBSD, native disk	39 MB/sec
μDenali NetBSD, interposed disk	37 MB/sec

Figure 10: **Disk interposition overhead.** This table shows the performance overhead of μDenali's virtual disks, with and without interposition.

due to interposition is *smallest* for short transfers. For example, the additional overhead for null interposition versus no interposition is 16% for 2k web requests and 41% for bulk TCP transfers. μDenali only imposes additional overhead on packet delivery, and therefore system call-intensive workloads are less affected by interposition. For Snort interposition, the per-packet overhead is large due to Snort's data copies and packet inspection routines. Snort is known to carry a high per-packet cost; even well-tuned installations can have difficulty keeping up with a heavily utilized 100 Mb/s Ethernet.

To evaluate disk performance, we used the UNIX dd utility to perform large, contiguous disk reads. In Figure 10, we compare the throughput of Linux, μDenali using a native (non-interposed disk), and μDenali using an interposed disk. The interposed disk is implemented as a file inside the parent VM's local file system. The parent's file system was initially empty, and therefore the child's blocks are likely to be nearly contiguous on disk. The sequential read workload is highly disk-bound, and therefore μDenali's numbers (with or without interposition) do not differ significantly from Linux.

6.3 Virtual Machine Services

We evaluated our Internet Suspend/Resume service using an Apache Web server as the migrating VM. We

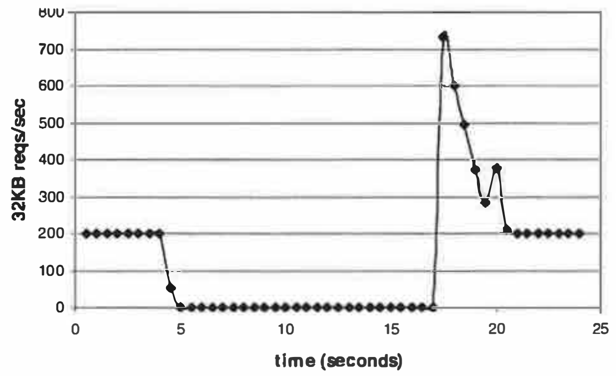


Figure 11: **Migrating an Apache server.** This timeline shows the behavior of an Apache Web service, before, during and after migration.

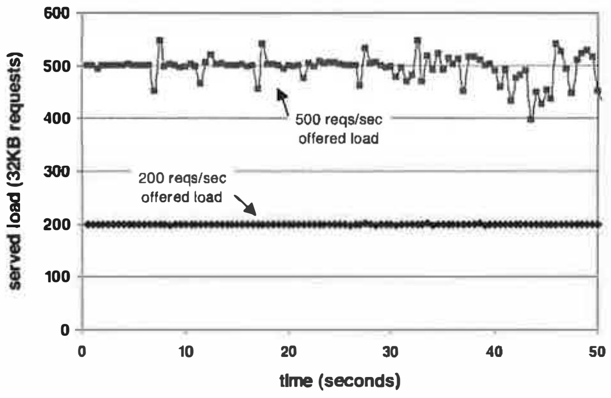


Figure 12: ***Apache** service performance.** This timeline shows the behavior of Apache before, during and after migration. The two lines correspond to two different workloads: 500 requests per second (open loop), and 200 requests per second (open loop).

configured a Web client to submit requests for a 32 kilobyte document 200 times per second. As shown in Figure 11, the Web server satisfies these requests before and after its migration. The instability that is evident immediately after migration is due to a batch of requests that are waiting inside the VM's Ethernet ring buffer. The long migration latency (12 seconds) is primarily due to our use of NFS as a transport mechanism.

We evaluated our *Apache** service using a three virtual machine Apache cluster. The parent VM was configured to garbage collect the active child VM after 10 seconds. Figure 12 shows the time-varying performance of *Apache** against offered loads of 200 requests per second and 500 requests per second for a 32KB Web document. For 200 requests per second, the throughput remains constant over time, which indicates no service disruption across VM restarts. At 500 requests per second, the system is less stable, suggesting the system is near its performance limit.

7 Related Work

μDenali is related to research into applications of virtual machines, novel virtual machine architectures, and extensible systems.

7.1 Virtual Machine Applications

Many researchers have been building novel applications that make use of virtual machine monitors. Internet Suspend/Resume [25] utilizes the checkpoint and resume functionality within VMware [35] to migrate virtual machines across a network. Sapuntzakis et al. [31] build a similar system, and show how to optimize performance with novel compression techniques. ReVirt [13] provides efficient logging and replay of virtual machines, and using this, King and Chen show how to identify sequences of events that lead to an intrusion [24]. The Stanford Collective project [30] uses VMMs to implement virtual appliances that facilitate software deployment and maintenance.

To implement these virtual machine services, the authors have had to reverse engineer relevant interfaces from a black-box VMM, or reimplement significant portions of the VMM in order to provide the interfaces they require. The goal of the μDenali VMM is to facilitate these kinds of services by exposing a carefully designed interposition and extension interface.

7.2 Novel Virtual Machine Monitors

Several research projects have focused on building novel virtual machine monitors. The Denali isolation kernel [36] relies on paravirtualization to implement lightweight virtual machines; μDenali is an enhancement to Denali. Xen [2] provides similar functionality to Denali, and focuses on providing high performance and strong isolation. Many user-level ports of Linux exist, including UMLinux [11]. Commercial virtual machine monitors have existed for several decades for mainframe computers [8], and have recently begun to achieve widespread usage on desktop workstations [35]. None of these virtual machine monitors provide comparable extensibility and interposition abilities to μDenali.

7.3 Extensibilty and Interposition

Numerous systems have injected novel functionality at the hardware interface. The storage device interface has been particularly fruitful. Petal virtual disks [26] offer a block-based 64-bit storage abstraction implemented on a cluster of workstations. Logical disks [9] provide an abstract disk interface based on logical block numbers and block lists, designed to support many different file system implementations. These systems do not attempt to be extensible or complete: they provide a fixed implementation of one virtual hardware device.

Other systems have explicitly dealt with adding extensibility to the hardware interface. User-mode pagers [7, 38] can be viewed as interposing on the implementation of the memory abstraction exposed to processes or operating systems. In particular, μDenali's virtual swap device closely resembles Mach's extensible paging for memory regions. μDenali moves beyond these systems to allow interposition on a much more complete spectrum of hardware abstractions.

Alpha PALCode [34] provides the ability to modify or extend the behavior of the Alpha instruction set architecture. Although this is a powerful primitive, PALCode faces several severe restrictions. PALcode is designed for small, low-level handlers such as TLB refills. PALCode is not suitable for implementing complex abstractions such as copy-on-write disks. A second limitation is that PALCode offers no abstraction: PALCode routines are architecture-dependent **and** implementation dependent. Finally, PAL instructions are differentiated from normal instructions in the instruction set, and therefore, it is impossible to interpose on arbitrary functionality.

The Java [19] virtual machine architecture is similar to μDenali at a high level, as it exposes a hardware-like architecture that is backed by a software-based implementation. As a result, Java VMs can support novel implementations, such as a distributed virtual machines [33]. However, JVMs are less complete than VMMs. They do not virtualize I/O devices such as Ethernet adapters or disks. As a result, JVMs do not support software systems with non-Java components. Additionally, applications such as migration that require the clean extraction of all I/O device state will be no easier to implement in a JVM than in a conventional OS.

μDenali's use of interposition as a means for achieving extensibility has been used by many previous systems, including interposition agents [23], Fluke's recursive virtual machines [16], and Mach's interposition enabled ports [12]. μDenali differs from previous systems in that we apply interposition to the virtual hardware interface exposed by a VMM. Thus, the class of applications and services enabled by μDenali is fundamentally different from previous systems, whose aim has been to interpose on the user/kernel boundary of a conventional operating system.

μDenali's architecture is similar to that of many microkernel-based operating systems, such as Mach [1] and L4 [21]. μDenali's port-based event routing framework is similar to the IPC mechanisms in these systems. Modern microkernels are able to run entire operating systems inside user-mode servers, including L^4linux [21]. μDenali differs from microkernels in many respects, providing the ability to extract full virtual hardware state, the ability to interpose on all virtual hardware events, and (currently in design only) the ability to log and reply non-deterministic events. In spirit, there is little difference between a microkernel and a VMM; both expose a virtual machine architecture. In practice, though, microkernels have been designed to provide extensible OS services such as user-mode pagers and file systems, whereas virtual machine monitors are being used to provide whole-machine services such as consolidation and migration.

There has been considerable past research into extensible operating systems [3, 14, 32]. SPIN [3] allows untrusted operating system extensions written in a type-safe language to be downloaded into the operating system. Unlike SPIN, we do not attempt to graft extensions into our kernel to override global system policies and mechanisms, but rather concentrate on extensions local to a single virtual machine. The Exokernel [14] allows programmers to build library operating systems that are safely multiplexed on a thin OS layer that exposes physical names. Instead of exposing physical names, μDenali exposes virtual names, sacrificing some performance to virtualization overhead in return for simplicity and potentially stronger isolation.

8 Conclusions and Future Directions

Virtual machine monitors have proven ideal for implementing a variety of system services, including migration, intrusion detection, and replay logging. All of these virtual machine services leverage the unique ability of a VMM to observe and interpose on the functionality and state of a complete software system. We believe there are many compelling applications of VMMs that are waiting to be discovered.

To facilitate these new services, we have redesigned the Denali VMM with the explicit goal of extensibility in mind. The resulting VMM, μDenali, has been architected to allow parent virtual machines to interpose functionality on behalf of their children. μDenali is structured on top of an event routing framework inspired by Mach ports. Using a high-level C library that exploits the extensibility features of μDenali, we implemented several virtual device extensions and virtual machine services that run in our framework, including Internet Suspend/Resume, a "drop-in" network intrusion detection virtual appliance, and a continuous rejuvenation framework for the Apache web server.

An exciting aspect of μDenali is that its extensibility mechanisms open up several avenues for future research. We are particularly interested in two future directions: using μDenali's checkpoint primitives to simplify the management of machines and networks, and using ReVirt-style logging to monitor for bugs like race conditions, which are otherwise difficult to analyze.

9 Acknowledgements

We are grateful for the feedback of our shepherd, Mendel Rosenblum, and our reviewers. We would also like to thank Brian Bershad, Ed Lazowska, and Hank Levy for their thoughts. Brian Youngstom's support in our lab was invaluable. This work was supported in part by NSF Career award ANI-0132817, funding from Intel Corporation, and a gift from Nortel Networks.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, 1986.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec 1995.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[7] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[8] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[9] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles (SOSP'93)*, Asheville, NC, December 1993.

[10] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[11] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[12] Richard Draves. A revised IPC interface. In *Proceedings of the USENIX Mach Conference*, October 1990.

13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

14] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

15] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

16] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Shantanu Goel, and Steven Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

17] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Tenth Annual Network and Distributed Systems Security Symposium*, San Diego, CA, February 2003.

19] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

20] Jim Gray. Why do computers stop and what can be done about it ? In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database systems*, January 1986.

21] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, October 1997.

22] Y. Huang, C. Kintala, and N. Kolettis. Software rejuvenation: analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

23] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, Asheville, NC, December 1993.

24] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, Bolton Landing, NY, October 2003.

25] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, June 2002.

[26] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[27] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, April 1989.

[28] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. In *Proceedings of the First Workshop on Internet Server Performance (WISP '98)*, Madison, WI, June 1998.

[29] J.S. Robin and C.E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX security symposium*, August 2000.

[30] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[31] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[32] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions, October 1996.

[33] Emin Gun Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, December 1999.

[34] Richard L. Sites. Alpha architecture reference manual, 1992.

[35] VMware, Inc. Vmware virtual machine technology. `http://www.vmware.com/`.

[36] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[37] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[38] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, November 1987.

SWAP: A Scheduler With Automatic Process Dependency Detection

Haoqiang Zheng and Jason Nieh
Department of Computer Science
Columbia University
{hzheng,nieh}@cs.columbia.edu

Abstract

We have developed SWAP, a system that automatically detects process dependencies and accounts for such dependencies in scheduling. SWAP uses system call history to determine possible resource dependencies among processes in an automatic and fully transparent fashion. Because some dependencies cannot be precisely determined, SWAP associates confidence levels with dependency information that are dynamically adjusted using feedback from process blocking behavior. SWAP can schedule processes using this imprecise dependency information in a manner that is compatible with existing scheduling mechanisms and ensures that actual scheduling behavior corresponds to the desired scheduling policy in the presence of process dependencies. We have implemented SWAP in Linux and measured its effectiveness on microbenchmarks and real applications. Our results show that SWAP has low overhead, effectively solves the priority inversion problem and can provide substantial improvements in system performance in scheduling processes with dependencies.

1 Introduction

Modern applications often consist of a number of cooperating processes in order to achieve a higher degree of modularity, concurrency, and performance. Applications of this type span a broad range from high-performance scientific parallel applications to desktop graphical computing applications. Interactions among the cooperating processes often result in dependencies such that a certain process cannot continue executing until some other processes finish certain tasks. However, operating systems today often ignore process dependencies and schedule processes independently. This can result in poor system performance due to the actual scheduling behavior contradicting the desired scheduling policy.

Consider priority scheduling, the most common form of scheduling used today in commercial operating systems for general-purpose and real-time embedded systems. The basic priority scheduling algorithm is simple: given a set of processes with assigned priorities, run the process with the highest priority. However, when processes share resources, resource dependencies among processes can arise that prevent the scheduler from running the highest priority process, resulting in priority inversion [5]. For example, suppose there are three processes with high, medium, and low priority such that the high priority process is blocked waiting for a resource held by the low priority process. A priority scheduler would run the medium priority process, preventing the low priority process from running to release the resource, thereby preventing the high priority process from running as well. This situation is particularly problematic because the medium priority process could run and prevent the high priority process from running for an unbounded amount of time. Priority inversion can critically impact system performance, as demonstrated in the case of the NASA Mars Pathfinder Rover [12] when priority inversion caused repeated system resets and drastically limited its ability to communicate back to the Earth.

Because priority inversion can cause significant performance problems, much work has been done to address this issue [1, 5, 15, 17]. The general idea behind these approaches is to boost the priority of a low priority process holding the resource so that it can run and release the resource to get out of the way of a high priority process waiting on the resource to run. However, there are four important limitations that occur in practice with such approaches in the context of general-purpose operating systems. First, these approaches focus on mutex resources only and do not address other potential resource dependencies among processes. For instance, a high priority X window application can suffer priority inversion while waiting on the X server to process requests from other lower priority X applications without any dependencies on mutexes [9]. Second, these approaches typically assume that it is possible to precisely determine dependencies among processes and do not consider dependencies such as those involving UNIX signals and System V IPC semaphores where no such direct correlation exists. Third, these approaches generally assume that priorities are static whereas priorities are often adjusted dynamically by scheduling policies in modern operating systems. Fourth, implementing these approaches for a given resource in a commercial operating system can require adding detailed resource-specific usage information and numerous modifications to many parts of a complex operating system.

We have developed *SWAP*, a *S*cheduler *W*ith *A*utomatic process de*P*endency detection, to effectively account for process dependencies in scheduling in the context of general-purpose operating systems. Rather than focusing on process dependencies arising from mutex resources, SWAP's dependency detection mechanism tracks system call history to determine a much broader range of possible resource dependencies among processes, including those that arise from widely used interprocess communication mechanisms. Because some dependencies cannot be precisely determined, SWAP associates a confidence level with each dependency that is dynamically adjusted using feedback from process blocking behavior. SWAP introduces a general dependency-driven scheduling mechanism that can use imprecise dependency information to schedule processes to run that are determined to be blocking high priority processes. SWAP scheduling is compatible with existing scheduling mechanisms. It is more general than popular priority inheritance approaches [5, 15] and can be used with schedulers that dynamically adjust priorities or non-priority schedulers. Furthermore, SWAP automatically accounts for process dependencies in scheduling without any intervention by application developers or end users. We have implemented SWAP in Linux and measured its effectiveness on both microbenchmarks and real applications. Our experimental results demonstrate that SWAP operates with low overhead and provides substantial improvements in system performance when scheduling processes with dependencies.

This paper presents the design and implementation of SWAP. Section 2 discuses related work. Section 3 describes the SWAP automatic dependency detection mechanism. Section 4 describes SWAP dependency-driven scheduling. Section 5 presents performance results that quantitatively measure the effectiveness of a Linux SWAP implementation using both microbenchmarks and real application workloads. Finally, we present some concluding remarks.

2 Related Work

Lampson and Redell discussed the priority inversion problem more than two decades ago and introduced priority inheritance to address the problem [5]. Using priority inheritance, a process holding a resource inherits the highest priority of any higher priority processes blocked waiting on the resource so that it can run, release the resource, and get out of the way of the higher priority processes. Priority inheritance assumes that priorities are static while they are inherited because recalculating the inherited priority due to dynamic priority changes is too complex. Priority inheritance addresses the priority inversion problem assuming the resource dependency is known; it does not address the underlying issue of determining resource dependencies.

Sha, et. al. developed priority ceilings [15] to reduce blocking time due to priority inversion and avoid deadlock in real-time systems. However, priority ceilings assume that the resources required by all processes are known in advance before the execution of any process starts. This assumption holds for some real-time embedded systems, but does not hold for general-purpose systems. Other approaches such as preemption ceilings [1] can also be used in real-time embedded systems but also make assumptions about system operation that do not hold for general-purpose systems. Like priority inheritance, priority ceilings typically assume static priorities to minimize overhead and does not address the issue of determining resource dependencies among processes.

To address priority inversion in the presence of dynamic priorities, Clark developed DASA for explicitly scheduling real-time processes by grouping them based on their dependencies [3]. While the explicit scheduling model is similar to our scheduling approach, DASA needs to know the amount of time that each process needs to run before its deadline in order to schedule processes. While such process information may be available in some real-time embedded systems, this information is generally not known for processes in general-purpose systems. DASA also assumes that accurate dependency information is provided. It does not consider how such information can be obtained, and can fail with inaccurate dependency information.

Sommer discusses the importance of removing priority inversion in general-purpose operating systems and identifies the need to go beyond previous work which focused almost exclusively on the priority inversion problem for mutex resources [17]. Sommer notes the difficulty of addressing priority inversion for non-mutex resources when it is difficult if not impossible to determine precisely on which process a high priority dependent process is waiting. Sommer proposes a priority-inheritance approach for addressing priority inversion due to system calls, but only implemented and evaluated an algorithm for a single local procedure system call in Windows NT. Sommer did not address general interprocess communication mechanisms that can result in priority inversion and does not consider the impact of dynamic priority adjustments.

Steere, et. al. developed a feedback-based resource reservation scheduler that monitors the progress of applications to guide resource allocation [18]. The scheduler allocates resources based on reservation percentages instead of priorities to avoid explicit priority inversion. Symbiotic interfaces were introduced to monitor application progress to derive the appropriate assignment of scheduling parameters for different applications based on their resource requirements in the presence of application dependencies. However, applications need to be modified to explicitly use the interfaces for the system to monitor progress effectively.

Mach's scheduler handoff mechanism [2], Lottery scheduling's ticket transfers [19] and doors [8] are mechanisms whereby applications can deal with process dependencies by explicitly having one process give its allocated time to run to another process. However, these handoff mechanisms typically require applications to be modified to explicitly use them. Applications also need to identify and know which processes to run. These mechanisms are not designed to resolve priority inversion in general and do not resolve priority inversions due to dependencies that are not explicitly identified in advance.

Co-scheduling mechanisms have been developed to improve the performance of parallel applications in parallel computing environments [4, 10, 16]. These mechanisms try to schedule cooperating processes or threads belonging to the same parallel application to run concurrently. This reduces busy waiting and context switching overhead and improves the degree of parallelism that can be used by the application. Because many of these applications are written using parallel programming libraries, these libraries can be modified to implement co-scheduling. Co-scheduling mechanisms focuses on supporting fine-grained parallel applications. They typically do not support multi-application dependencies and do not address the problem of uniprocessor scheduling in the presence of process dependencies.

3 Automatic Dependency Detection

SWAP introduces a mechanism that automatically detects potential process dependencies by leveraging the control flow structure of commodity operating systems. In commodity operating systems such as Linux, process dependencies occur when two processes interact with each other via the interprocess communication and synchronization mechanisms provided by the operating system. These mechanisms are provided by the operating system as system calls. This suggests a simple idea that SWAP uses for detecting resource dependencies among processes: if a process is blocked because of a process dependency, determine the system call it was executing and use that information to determine what resource the process is waiting on and what processes might be holding the given resource.

Based on this idea, SWAP uses a simple resource model to represent process dependencies in a system. The model contains three components: resources, resource requesters, and resource providers. A resource requester is simply a process that is requesting a resource. A resource provider is a process that may be holding the requested resource and therefore can provide the resource by releasing it. If a certain resource is requested but is not available, the resource requesters will typically need to block until the resource is made available by the resource providers. SWAP uses this simple yet powerful model to represent almost all possible dependency relationships among processes. SWAP applies this model to operating system resources to determine dependencies resulting from interprocess communication and synchronization mechanisms.

An assumption made by SWAP is that resources are accessed via system calls. While this is true for many resources, one exception is the use of memory values for synchronization, most notably user space shared memory mutexes. User space mutexes may simply spin wait to synchronize access to protected resources. Since no system call is involved when accessing this kind of mutex, SWAP does not detect this kind of dependency relationship. However, thread library mutex implementations such as pthreads in Linux do not allow spin waiting indefinitely while waiting for a mutex. Instead, they allow spin waiting for only a time less than the context switch overhead then block. Given that context switch times in modern systems are no more than a few microseconds, the time spent busy waiting and the time not accounted for by SWAP is relatively small. For example, the Linux pthread mutex implementation will spin wait for only 50 CPU cycles before blocking [6]. SWAP focuses instead on process dependencies that can result in processes blocking for long periods of time.

3.1 SWAP Resource Model

In the SWAP resource model, each resource has a corresponding resource object identified by a tuple consisting of the resource type and the resource identifier. The resource identifier can consist of an arbitrary number of integers. The meaning of the resource identifier is specific to the type of resource. For example, a socket is a resource type and the inode number associated with this socket object is used as the resource specific identifier for sockets. SWAP associates with each resource object a list of resource requesters and a list of resource providers.

SWAP creates a resource object when a resource is accessed for the first time and deletes the object when there are no more processes using it, which is when it has no more resource requesters or providers. SWAP efficiently keeps track of resource objects by using a resource object hash table. A resource object is added to the hash table when it is created and removed when it is deleted. The hash key of a resource object is generated from its resource identifier. If a resource identifier consists of N integers, SWAP uses the modulus of the sum of all these integers and the hash table size as the hash key for this resource. Generating the hash key this way allows resource objects to be quickly added into or retrieved from the hash table. Separate chaining is used if resource identifiers hash to the same hash table entry, but such conflicts are infrequent.

SWAP associates a process as a resource requester for a resource object if the process blocks because it is requesting the respective resource. When a process blocks, SWAP needs to first determine what resource the process is requesting. Since resources are accessed via system calls, SWAP can determine the resource being requested by examining the system call parameters. For example, if a process requests data from a socket resource by using the system call `read(sock,...)`, we can identify which socket this process is accessing from the socket descriptor `sock`. When a process executes a system call, SWAP saves the parameters of the system call. If the process blocks, SWAP then identifies the resource being requested based on the saved system call parameters. Once the resource is identified, SWAP appends the process to the resource object's list of resource requesters. When a resource requester eventually runs and completes its system call, SWAP determines that its request has been fulfilled and removes the process from the requester list of the respective resource object.

To allow the resource requester to wake up and continue to run, a resource provider needs to run to provide the respective resource to the requester. To reduce the time the resource requester is blocked, we would like to schedule the resource provider as soon as possible. However, waking up the requester by providing the necessary resource is an action that will not happen until some time in the future. Knowing which process will provide the resource to wake up the requester is unfortunately difficult to determine before the wake up action actually occurs. The process that will provide the resource may not have even been created yet. Furthermore, there may be multiple processes that could serve as the provider for the given requester. For example, a process that is blocked on an IPC semaphore could be provided the semaphore by any process in the system that knows the corresponding IPC key. That is to say, any process existing in the system could in theory be the possible resource provider. In practice though, only a few processes will have the corresponding IPC key and hence the number of possible resource providers may be more than one but is likely to be small.

To identify resource providers, SWAP uses a history-based prediction model. The model is based on the observation that operating system resources are often accessed in a repeating pattern. For example, once a process opens a socket, it will usually make many calls to access the socket before having it closed. This behavior suggests that a process with a history of being a good resource provider is likely to be a future provider of this resource. SWAP therefore treats all past providers of a certain resource as potential future resource providers. SWAP first identifies these potential resource providers and then applies a feedback-based confidence evaluation using provider history to determine which potential providers will actually provide the necessary resource to a requester in the most expedient manner.

SWAP associates a process as a potential resource provider for a resource object the first time the process executes a system call that makes it possible for the process to act as a resource provider. For example, if a process writes data to a socket resource, SWAP identifies this process as a resource provider for the socket. When a process executes a system call, SWAP determines the resource being provided by examining the system call parameters. Once the resource is identified, SWAP appends the process to the resource object's list of resource providers. Note that when SWAP adds a process to the resource provider list, it has only been identified as a potential resource provider. The potential provider must have provided the resource at least once to be identified as a resource provider, but just because it has provided the resource before does not necessarily mean it will provide the resource again.

SWAP uses feedback-based confidence evaluation to predict which process in a list of potential resource providers will provide the necessary resource most quickly to a resource requester. This is quantified by assigning a confidence value to each potential resource provider. A larger confidence value indicates that a provider is more likely to provide the resource quickly to a resource requester. SWAP adjusts the confidence values of potential providers based on feedback from their ability to provide the resource quickly to a requester. If a resource provider is run and it successfully provides the resource to a requester, SWAP will use that positive feedback to increase the confidence value of the provider. If a resource provider is run for a certain amount of time and it does not provide the resource to a requester, SWAP will use that negative feedback to decrease the confidence value of the provider.

We first describe more precisely how SWAP computes the confidence of each resource provider. Section 4 describes how SWAP uses the confidence of resource providers in scheduling to account for process dependencies. SWAP assigns an initial base confidence value K to a process when it is added to a resource object's provider list. SWAP adjusts the confidence value based on feedback within a range from 0 to $2K$. K can be configured on a per resource basis. If a resource provider successfully provides the resource to a requester, SWAP increments its confidence value by one. If a resource provider runs for T time quanta and does not provide the resource to a requester, SWAP decrements its confidence value by one. T can be configured on a per resource basis. A process on the resource provider list will not be considered as a potential resource provider if its confidence drops to zero.

Because it is possible to have cascading process dependencies, a resource provider P_1 for a resource requester P can further be blocked by another process P_2, which is the resource provider for P_1. As a result, P_2 can be indirectly considered as a resource provider for P. SWAP dynamically determines the confidence of an indirect provider as the product of the respective confidence values. Let $C(P, P_1, R)$ be the confidence of provider P_1 for resource R with requester P and $C(P_1, P_2, R_1)$ be the confidence of provider P_2 for resource R_1 with requester P_1. Then the indirect confidence $C(P, P_2, R)$ is computed as $C(P, P_1, R) * C(P_1, P_2, R_1)/K$. Since P_2 is not a direct resource provider for P, if P_2 is run as an indirect resource provider for P, the feedback from that experience is applied to the confidence of P_1. This ensures that a resource provider that is blocked and has multiple providers itself will not be unfairly favored by SWAP in selecting among direct resource providers based on confidence values.

A process will usually remain on the resource provider list until it either terminates or executes a system call that implicitly indicates that the process will no longer provide the resource. For example, a process that closes a socket would no longer be identified as a resource provider for the socket. SWAP does, however, provide a configurable parameter L that limits the number of resource providers associated with any resource object. SWAP groups the providers in three categories: high confidence providers, default confidence providers, and low confidence providers. When this parameter L is set, SWAP only keeps the L providers with highest confidence values. If a new resource provider needs to be added to the provider list and it already has L providers, the provider is added to the end of the default confidence provider list, and an existing provider is removed from the front of the lowest category provider list that is not empty. When there are many potential resource providers, this limit can result in a loss of past history information regarding low confidence providers, but reduces the history information maintained for a resource object.

3.2 SWAP Dependency Detection in Linux

To further clarify how the SWAP resource model can be generally and simply applied to automatically detecting process dependencies in general-purpose operating systems, we consider specifically how the model can be applied to the kinds of resources found in Linux. These resources include sockets, pipes and FIFOs, IPC message queues and semaphores, file locks, and signals. We discuss in detail how sockets, IPC semaphores, and file locks can be identified by SWAP and how the requesters and potential providers of these resources can be automatically detected. Pipes, FIFOs, IPC message queues and signals are supported in a similar fashion but are not discussed further here due to space constraints.

Sockets are duplex communication channels that can involve communication either within a machine or across machines. We only consider the former case since the SWAP resource model only addresses process dependencies within a machine. This includes both UNIX domain sockets and Internet sockets with the same local source and destination address. A socket has two endpoints and involves two processes, which we can refer to as a server and a client. SWAP considers each endpoint as a separate resource so that each socket has two peer resource objects associated with it. These objects can be created when a socket connection is established. For example, when a client calls `connect` and a server calls `accept` to establish a socket connection, SWAP creates a client socket resource object and a server resource object. All system calls that establish and access a socket provide the socket file descriptor as a parameter. SWAP can use the file descriptor to determine the corresponding inode number. SWAP then uses the inode number to identify a socket resource object.

SWAP determines the resource requesters and providers for socket resource objects based on the use of system calls to access sockets. A socket can be accessed using the following system calls: `read`, `write`, `readv`, `writev`, `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, `recvmsg`, `select`, `poll`, and `sendfile`. When a socket is accessed by a process, the process is added as a resource provider for its socket endpoint and the system call parameters are saved. The process will remain a resource provider for the resource object until it explicitly closes the socket or terminates. If the system call blocks, it means that the process is requesting the peer resource object on the other end of the socket. For example, when a client does a `read` on its endpoint and blocks, it is because the server has not done a `write` on its peer endpoint to make the necessary data available to the client. If the system call blocks, SWAP therefore adds the process as a resource requester for the peer resource object.

System V IPC semaphores provide an interprocess synchronization mechanism. SWAP associates a resource object with each IPC semaphore. An IPC semaphore object is created when a process calls `semget` to create a semaphore. `semget` returns a semaphore identifier which is a parameter used by all system calls that access the semaphore. SWAP therefore uses the semaphore identifier to identify the respective resource object. SWAP determines the resource requesters and providers for semaphores based on the use of the `semop` system call to access the resource. When a semaphore is accessed by a process, the process is added as a resource provider and the system call parameters are saved. The process will remain a resource provider for the resource object until the semaphore is explicitly destroyed or the process terminates. If the system call blocks, the calling process is added as a resource requester of the semaphore object.

File locks provide a file system synchronization mechanism between processes. Linux supports two kinds of file locking mechanisms, fcntl and flock. Since both of these mechanisms work in a similar way and can both provide reader-writer lock functionality, we just discuss how SWAP supports the flock mechanism. SWAP associates a resource object with each file lock. An flock object is created when a process calls `flock` to create a file lock for a file associated with the file descriptor parameter in `flock`. SWAP distinguishes between exclusive locks and shared locks and creates a different flock resource object for each case. Since `flock` is used for all operations on the file lock, the file descriptor is available for all file lock operations. SWAP can therefore use the file descriptor to determine the corresponding inode number. SWAP uses the inode number and a binary value indicating whether the file lock created is shared or exclusive to identify the respective resource object.

SWAP determines the resource requesters and providers for flock resource objects based on the use of system calls to access the resources. A shared flock resource object is accessed when `flock` is called with `LOCK_SH` and the respective file descriptor. When this happens, the calling process is added as a resource provider for the object. If the system call blocks, then some other process is holding the file lock exclusively, which means the process is a resource requester for the exclusive flock object. SWAP therefore adds the process as a resource requester for the exclusive flock resource object. An exclusive flock resource object is accessed when `flock` is called with `LOCK_EX` and the respective file descriptor. When this happens, the calling process is added as a resource provider for the object. If the system call blocks, then some other process is holding the file lock either shared or exclusively, which means the process is a resource requester for the both the exclusive and shared flock object. SWAP therefore adds the process as a resource requester for both flock resource objects. For both shared and exclusive flock resource objects, a process remains a resource provider for the object until it terminates or explicitly unlocks the flock by calling `flock` with `LOCK_UN`.

4 Dependency-Driven Scheduling

SWAP combines the information it has gathered from its dependency detection mechanism with a scheduling mechanism that is compatible with the existing scheduling framework of an operating system but accounts for process dependencies in determining which process to run. We describe briefly how a conventional scheduler determines which process to run and then discuss how SWAP augments that decision to account for process dependencies.

A conventional scheduler maintains a run queue of runnable processes and applies an algorithm to select a process from the run queue to run for a time quantum. Processes that block become not runnable and are removed from the run queue. Similarly, processes that wake up become runnable and are inserted into the run queue. With no loss of generality, we can view each scheduling decision as applying a priority function to the run queue that sorts the runnable processes in priority order and then selects the highest priority process to execute [14]. This priority model, where the priority of a process can change dynamically for each scheduling decision, can be used for any scheduling algorithm. To account for process dependencies, we would like the priority model to account for these dependencies in determining the priority of each process. Assuming the dependencies are known, this is relatively easy to do in the case of a static priority scheduler where each process is assigned a static priority value. In this case when a high priority process blocks waiting on a lower priority process to provide a resource, the the lower priority process can have its priority boosted by priority inheritance. The scheduler can then select a process to run based on the inherited priorities. However, priority inheritance is limited to static priority schedulers and does not work for more complex, dynamic priority functions.

To provide a dependency mechanism that works for all dynamic priority functions and therefore all scheduling algorithms, SWAP introduces the notion of a *virtual runnable* process. A virtual runnable process is a resource requester process that is blocked but has at least one runnable resource provider or virtual runnable resource provider that is not itself. Note that a resource requester could potentially also be listed as a provider for the resource, but is explicitly excluded from consideration when it is the resource requester. The definition is recursive in that a process's provider can also be virtual runnable. A process that is blocked and has no resource providers is not virtual runnable. A virtual runnable process can be viewed as the root of a tree of runnable and virtual runnable resource providers such that at least one process in the tree is runnable. SWAP makes a small change to the conventional scheduler model by leaving virtual runnable processes on the run queue to be considered in the scheduling decision in the same manner as all other runnable processes. If a virtual runnable process is selected by the scheduler to run, one of its resource providers is instead chosen to run in its place. SWAP selects a resource provider to run in place of a virtual runnable process using the confidence associated with each provider. The confidence of a runnable provider is just its confidence value. The confidence of a virtual runnable provider is its indirect confidence value as described in Section 3.1. If a virtual runnable provider is selected, the confidence values of its providers are examined recursively until a runnable process with the highest confidence value is selected. Once a virtual runnable requester process becomes runnable as a result of being provided the resource, it is simply considered in the scheduling decision like all other runnable processes.

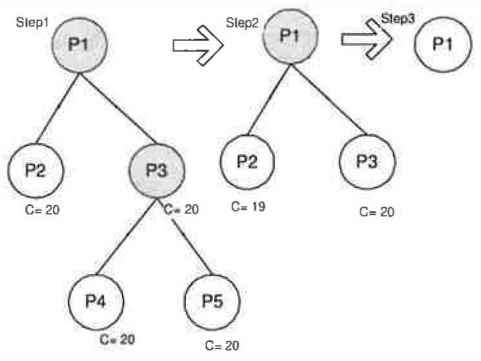


Figure 1: SWAP Scheduling Example

Figure 1 shows an example to further illustrate how SWAP dependency-driven scheduling chooses a resource provider based on confidence. Virtual runnable processes are shaded and runnable processes are not. Suppose at the time the scheduler is called, a virtual runnable process P1 is currently the highest priority process. In Step 1 in Figure 1, P1 is blocked because it is requesting resource R1 which has two resource providers P2 and P3. P3 is also blocked because of it is requesting resource R2 which has two providers P4 and P5. In this example, P4 needs to run for 40 ms to produce resource R2, and P3 needs to run for 20 ms to produce resource R1. P2 and P5 do not actually provide the respective resources this time. In this example, we assume the confidence quantum T described in Section 3.1 is 100 ms. SWAP needs to decide which process among P2, P4, and P5 is likely to wake up P1 early. It decides by using the confidence value associated with each provider. Assume there is no previous confidence history for these resources so that all confidence values are equal to a base confidence K of 20. In deciding which process to run in place of virtual runnable process P1, SWAP then determines P2's confidence $C(P1, P2, R1)$ as 20, P4's indirect confidence $C(P1, P4, R1)$ as $C(P1, P3, R1) * C(P3, P4, R2)/K = 20$, and P5's indirect confidence $C(P1, P5, R1)$ as $C(P1, P3, R1)*C(P3, P5, R2)/K = 20$. The first provider with the highest confidence value is P2, so it will be selected to run first. It will run for 100 ms and then receive negative feedback because it does not wake up P1 after one confidence time quantum is used. $C(P1, P2, R1)$ will then become 19. P4 becomes the first provider with the highest confidence, so it will be selected to run. After P4 runs for 40 ms, it provides the resource for P3, which wakes up P3, resulting in the new situation shown in Step 2 in Figure 1. At this time, P3 can be the selected to run for virtual runnable P1 since it has a higher confidence value than P2. It will continue to run for another 20 ms and eventually wake up P1, resulting in P1 being runnable as shown in Step 3 in Figure 1.

SWAP's use of virtual runnable processes provides a number of important benefits. By introducing a new virtual runnable state for processes, SWAP leverages an existing scheduler's native decision making mechanism to account for process dependencies in scheduling. SWAP implicitly uses an existing scheduler's dynamic priority function without needing to explicitly calculate process priorities which could be quite complex. SWAP does not need to be aware of the scheduling algorithm used by the scheduler and does not need to replicate the algorithm as part of its model in any way. As a result, SWAP can be integrated with existing schedulers with minimal scheduler changes and can be easily used with any scheduler algorithm, including commonly used dynamic priority schemes. By using a confidence model, SWAP allows a scheduler to account for process dependency information in scheduling even if such information is not precisely known.

One issue with using the confidence model in scheduling is that it may unfairly favor processes with high confidence. If SWAP keeps selecting a given provider, its confidence will continue to rise if it never fails to provide the necessary resource. This behavior is consistent with SWAP's objective to run the process that can most quickly provide the resource for a virtual runnable process. However, there may be other providers that could also provide the resource just as quickly but are not selected by the confidence model because they were not initially given the chance to run. We have not seen this issue in practice for three reasons. First, the process that runs in place of a virtual runnable process is still charged by the underlying scheduler for the time it runs, so running it sooner results in it running less later. Second, a process that provides a resource typically only runs for a short duration until it provides the resource. A process selected using the confidence model also must provide the resource within a confidence quantum of T which is usually small, otherwise other resource providers will be selected. Third, the confidence model is only used when the resource requester blocks because the resource is not available and there are multiple potential providers. If a requester does not need to block or there is only one provider, the confidence model is not used.

While the SWAP approach provides important advantages, we also note that it can be limited by the decision making algorithm of an existing scheduler in the context of multiprocessors. To support scalable multiprocessor systems, schedulers typically associate a separate run queue with each CPU to avoid lock contention on a centralized run queue. Since CPU scheduling decisions are decoupled from one another, the process that is selected to run may not be the most optimal. In a similar manner, if a virtual runnable process is selected by the scheduler on a CPU, it may not be globally the best virtual runnable process to select. It is possible that the virtual runnable process selected is not the one with the highest priority across all CPUs according to the native scheduler's priority model. Other approaches could be used to select a globally more optimal virtual runnable process, but would incur other disadvantages. One could provide separately managed queues for virtual runnable processes, but this would require duplicating scheduler functionality and increasing complexity. If a single queue was used for virtual runnable processes, this could also impact scheduler scalability.

5 Experimental Results

We have implemented a SWAP prototype in Red Hat Linux 8.0 which runs the Linux 2.4.18-14 kernel. The SWAP automatic dependency detection mechanisms were designed in such a way that they can be implemented as a loadable kernel module that does not require any changes to the kernel. The SWAP dependency-driven scheduling mechanism was also largely implemented in the same kernel module, but it does require some changes to the kernel scheduler. These changes only involved adding about 15 lines of code to the kernel and were localized to only a couple of kernel source code files related to the kernel scheduler. As discussed in Section 3.1, SWAP provides three configurable parameters, the default maximum number of providers per resource L, the initial base confidence value K, and the confidence quantum T. In our SWAP implementation, the default values of L, K, and T were set to 20, 20, and 20 ms, respectively.

We have used our SWAP prototype implementation in Linux to evaluate its effectiveness in improving system performance in the presence of process dependencies. We compared Linux SWAP versus vanilla Redhat Linux 8.0 using both microbenchmarks and real client-server applications. Almost all of our measurements were performed on an IBM Netfinity 4500R server with a 933 MHz Intel PIII CPU, 512 MB RAM, and 100 Mbps Ethernet. We also report measurements obtained on the same machine configured with two CPUs enabled. We present some experimental data from measurements on four types of application workloads. Section 5.1 presents results using a client-server microbenchmark workload to measure the overhead of SWAP and illustrate its performance for different resource dependencies. Section 5.2 presents results using a multi-server microbenchmark to measure the effectiveness of SWAP when multiple processes can be run to resolve a resource dependency. Section 5.3 presents results using a thin-client computing server workload to measure the effectiveness of SWAP in a server environment supporting multiple user sessions. Section 5.4 presents results using a chat server workload to measure the effectiveness of SWAP in a multiprocessor server environment supporting many chat clients.

5.1 Client-server Microbenchmark

The client-server microbenchmark workload consisted of a simple client application and server application that are synchronized to start at the same time. Both the client and the server run in a loop of 100 iterations. In each iteration, the client waits for the server to perform a simple bubblesort computation on a 4K array and respond to the client via some method of communication, resulting in a dependency between client and server. We considered six common communication mechanisms between client and server:

- Socket (SOCK): Server computes and writes a 4 KB data buffer to a Unix domain socket. Client reads from the socket.
- Pipe/FIFO (PIPE): Server computes and writes a 4 KB data buffer to a pipe. Client reads the data from the pipe.
- IPC message queue (MSG): Server computes and sends a 4 KB data buffer via an IPC message queue. Client receives the data from the message queue.
- IPC semaphores (SEM): Two semaphores called *empty* and *full* are initialized to true and false, respectively. Server waits for *empty* to be true then computes and signals *full* when it completes. Client waits for *full* to be true then signals *empty*. Wait and signal are implemented using the `semop` system call.
- Signal (SIG): Server waits for a signal from the client, computes, and sends a signal to client when it completes its computation. Client waits until it receives the signal.
- File locking (FLOCK): Server uses `flock` to lock a file descriptor while it does its computation and unlocks when it is completed. Client uses `flock` to lock the same file and therefore must wait until server releases the lock.

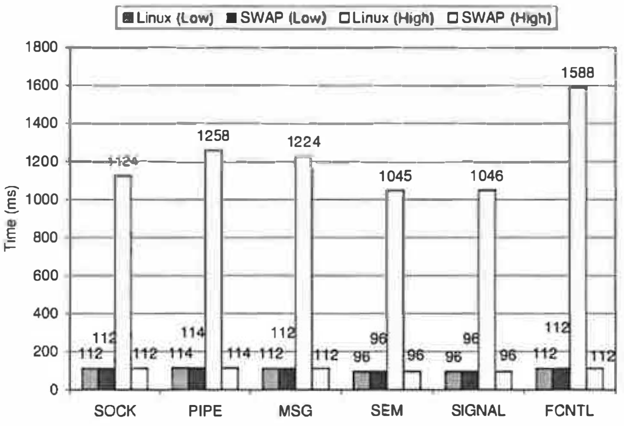


Figure 2: Client-server Microbenchmark Results

We measured the average time it took the client to complete an iteration of each of the six client-server microbenchmarks when using vanilla Linux versus SWAP. For this experiment, we assumed that the client is an important application and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes are run using the default SCHED_-OTHER scheduling policy. In Linux, SCHED_FIFO processes are higher priority than SCHED_OTHER processes and are therefore scheduled to run before SCHED_OTHER processes. We measured the client iteration completion time when there were no other applications running on the system to further compare the overhead of SWAP with vanilla Linux. We then measured the client iteration completion time using vanilla Linux versus SWAP when ten other application processes were running at the same time as the client-server microbenchmark. The application processes were simple while loops imposing additional load on the system. This provides a measure of the performance of vanilla Linux versus SWAP on a loaded system in the presence of process dependencies.

Figure 2 shows the measurements for each of the six client-server microbenchmarks. For low system load, the measurements show that the client iteration completion time for each microbenchmark was roughly 100 ms for both vanilla Linux and SWAP. The client completed quickly in all cases, and the difference between the completion times using SWAP and vanilla Linux were negligible. Figure 3 shows the average total time of executing system calls in each iteration for the six microbenchmarks. This provides a more precise measure of the overhead of SWAP versus vanilla Linux. The extra system call overhead of SWAP ranges from 1.3 μs for the FLOCK microbenchmark to 3.8 μs for the SEM microbenchmark. Although the absolute processing time overhead for SWAP is smallest for the FLOCK microbenchmark, it is the largest percentage overhead of all the microbenchmarks because the overall system call cost of the FLOCK microbenchmark is small.

The extra overhead associated with SWAP is due to the costs of intercepting the necessary system calls for each microbenchmark, examining their parameters, determining which if any processes should be added or removed from the SWAP resource requester and provider lists, and performing dependency driven scheduling when necessary. The difference in SWAP overhead for the different microbenchmarks is due to different numbers of SWAP operations being done in each case. For each iteration of the FLOCK microbenchmark, there are two context switches, resulting in extra SWAP scheduling overhead since it needs to schedule virtual runnable processes. In addition for each iteration, SWAP intercepts two system calls and manages two resource objects for the file lock, one corresponding to exclusive lock access and the other corresponding to shared lock access. For these two system calls, SWAP needs to check the status of resource providers and requesters a total of three times. For each iteration of the SEM microbenchmark, there are also two context switches, resulting in extra SWAP scheduling overhead since it needs to schedule virtual runnable processes. In addition for each iteration, SWAP intercepts four system calls and manages two resource objects corresponding to two IPC semaphores used by the SEM microbenchmark. For these system calls, SWAP needs to check the status of resource providers and requesters a total of eight times. The higher number of resource provider and requester operations for the SEM microbenchmark results in the higher SWAP processing costs compared to the FLOCK microbenchmark. While these microbenchmarks were designed to exercise those system calls that SWAP intercepts, note that SWAP only intercepts a small minority of system calls and only the intercepted system calls incur any additional overhead.

While Figure 3 shows the average overhead due to system calls across all iterations of the microbenchmarks, the cost of a system call can be larger the first time it is called when using SWAP due to extra memory allocation that may occur to create resource objects, resource providers, and resource requesters. For each resource, SWAP creates a 64 byte resource object. When a process is first added as a resource requester for a resource, SWAP creates a 32 byte resource requester object. When a process is first added as a resource provider for a resource, SWAP creates a 32 byte resource provider

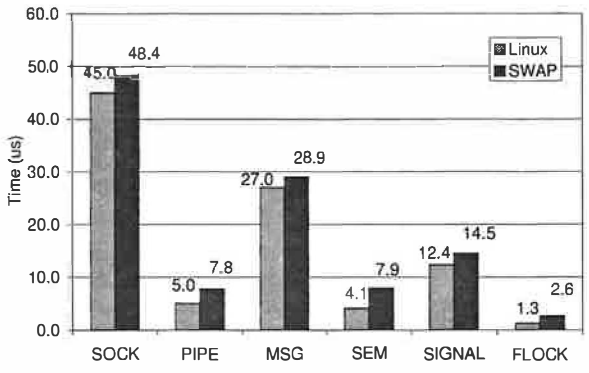


Figure 3: SWAP Overhead

object. The memory overhead of SWAP grows with the number of resource objects, providers, and requesters that need to be allocated. However, since the associated SWAP objects are all small, and the resulting memory overhead imposed by SWAP ends up also being small.

For high system load, Figure 2 shows that the client iteration completion time for each microbenchmark was an order of magnitude better using SWAP versus vanilla Linux. Despite the fact that the client was the highest priority process in the system, the client iteration completion times using vanilla Linux ballooned to over 1 second, roughly ten times worse than for low system load. The problem is that the client depends on the server, which runs at the same default priority as the other processes in the system. Since Linux schedules processes independently, it does not account for the dependency between client and server, resulting in the high priority client process not being able to run. In contrast, the client iteration completion times when using SWAP remained almost the same for both high and low system load at roughly 100 ms for all of the microbenchmarks. The client performance using SWAP for high system load is roughly ten times better than vanilla Linux for high system load and essentially the same as vanilla Linux for low system load. SWAP automatically identifies the dependencies between client and server processes for each microbenchmark and correctly runs the server process ahead of other processes when the high priority client process depends on it.

5.2 Multi-server Microbenchmark

The multi-server microbenchmark workload consisted of a simple client application and five server applications that are started at the same time. The microbenchmark is similar to the SEM microbenchmark described in Section 5.1 with two differences. First, since each server increments the semaphore and there are multiple servers running, the client will only need to wait until one of the servers increments the semaphore before it can run and decrement the semaphore. Second, each of the servers may do a different number of bubblesort computations, resulting in the server processing taking different amounts of time. For this experiment, the five servers repeated the bubblesort computation 2, 5, 5, 10, and 10 times, respectively. As a result, the servers vary in terms of the amount of processing time required before the semaphore is incremented.

We measured the time for the client to complete each of the first 15 loop iterations when using vanilla Linux versus SWAP. For SWAP, we considered the impact of different confidence feedback intervals by using two different intervals, 20 ms and 200 ms. For this experiment, we assumed that the client is an important application and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes in the system are run using the default SCHED_OTHER scheduling policy. We measured the client iteration time when no other applications were running on the system as a baseline performance measure on vanilla Linux and SWAP. We then measured the client completion time using vanilla Linux versus SWAP when ten other application processes were running at the same time as the client-server microbenchmark. The application processes were simple while loops imposing additional load on the system. This provides a measure of the performance of vanilla Linux versus SWAP on a loaded system in the presence of process dependencies.

Figure 4 shows the multi-server microbenchmark measurements. It shows the measured client iteration completion time for each iteration using vanilla Linux and SWAP for both low system load and high system load. In this figure, SWAP-20 is used to denote the measurements done with a SWAP confidence feedback interval of 20 ms and SWAP-200 is used to denote the measurements done with a SWAP confidence feedback interval of 200 ms.

For low system load, Figure 4 shows that client iteration time is roughly the same at 1 second when using SWAP or vanilla Linux for the first iteration. However, the client iteration time when using SWAP is much better than when using vanilla Linux for subsequent iterations. While the client iteration time remains at roughly 1 second for all iterations when using vanilla Linux, the client iteration time drops to about 200 ms when using SWAP, with the iteration time dropping faster using SWAP-200 versus SWAP-20. Of the five servers running, the server running the bubblesort computation twice increments

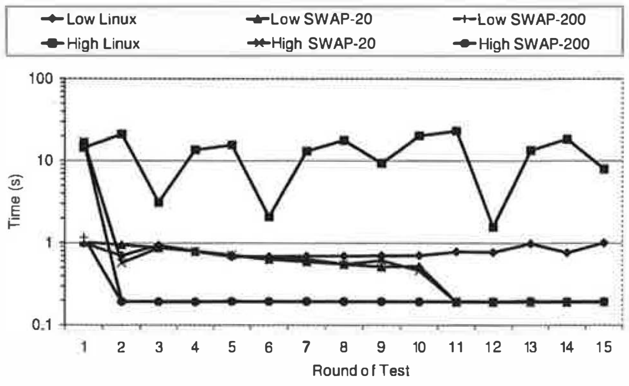


Figure 4: Multi-server Microbenchmark Results

the semaphore the fastest at roughly 200 ms. As a result, the client completes its iteration the fastest when this server is scheduled to run instead of the other servers. The results in Figure 4 show that SWAP eventually finds the fastest server to run to resolve the process dependency between the high priority client and the servers.

It takes the client about 1 second to complete an iteration using vanilla Linux because the default SCHED_OTHER scheduling policy used in the Linux 2.4.18-14 kernel is basically a round-robin scheduling algorithm with a time quantum of 150 ms. The Linux scheduler therefore will end up running each of the five servers in round-robin order until one of the servers increments the semaphore allowing the high-priority client to run and complete an iteration. The fastest server needs to run for 200 ms to increment the semaphore. Therefore, depending on when the fastest server is run in round-robin order, it could take between 800 ms to 1400 ms until the semaphore is incremented and the client can run, which is consistent with the results shown in Figure 4.

On the other hand, using SWAP the client only takes 200 ms to complete an iteration because SWAP's confidence feedback model identifies the fastest server after the first iteration because that server is the one that increments the semaphore and allows the client to run. In subsequent iterations, SWAP gives that server preference to run, resulting in lower client iteration time. Figure 4 also shows that SWAP with a feedback interval of 200 ms will reach the optimal level faster than SWAP with a feedback interval of 20 ms. This is because the confidence value is adjusted one unit for each feedback, which means each positive feedback will make the process run 1 quantum more ahead of the other processes. The larger the quantum, the more benefit a process will receive from a positive feedback. In this sense, it is desirable for the confidence quantum to be as large as possible. However, if the quantum is too large, the dependency-driven scheduler will behave in a FIFO manner, which can result in longer response times. In this case, configuring the confidence time quantum to be 200 ms works well because it is the time needed for the fastest provider to produce the desired resource.

For high system load, Figure 4 shows that client iteration time is roughly the same at 10 seconds when using SWAP or vanilla Linux for the first iteration. However, the client iteration time when using SWAP is significantly better than when using vanilla Linux for subsequent iterations. The reasons for SWAP's better performance for high system load are the same as for low system load, except that the difference between SWAP and vanilla Linux is magnified by the load on the system.

5.3 Thin-client Computing Server

The thin-client computing server workload consisted of VNC 3.3.3 thin-client computing sessions running MPEG video players. VNC [13] is a popular thin-client system in which applications are run on the server and display updates are then sent to a remote client. Each session is a complete desktop computing environment. We considered two different VNC sessions:

- MPEG play: The VNC session ran the Berkeley MPEG video player [11] which displayed a locally stored 5.36 MB MPEG1 video clip with 834 352x240 pixel video frames.
- Netscape: The VNC session ran a Netscape 4.79 Communicator and downloaded and displayed a Javascript-controlled sequence of 54 web pages from a web server. The web server was a Micron Client Pro with a 450 MHz Intel PII, 128 MB RAM, and 100 Mbps Ethernet, running Microsoft Windows NT 4.0 Server SP6a and Internet Information Server 3.0.

We measured the time it took for the respective application in one VNC session to complete when using vanilla Linux versus SWAP. For this experiment, we assumed that the application measured, either the video player or web browser, is important and is therefore run as a real-time SCHED_FIFO process in Linux. All other processes in the system are run using the default SCHED_OTHER scheduling policy. We measured the respective video player and web browser completion times when there were no other applications running on the system to provide a baseline performance measure of each application running on vanilla Linux and SWAP. We then measured each application completion time using vanilla Linux versus SWAP with 50 other VNC sessions running at the same time, each session running the video player application.

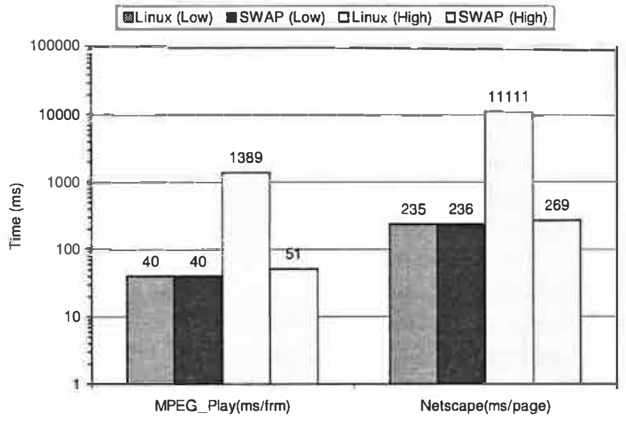


Figure 5: Thin-client Computing Server Benchmark Results

Figure 5 shows the thin-client computing server measurements for the VNC session running MPEG play and the VNC session running Netscape. For low system load, the measurements show that the client completion time for both real applications was roughly the same. The overhead caused by SWAP is less than 0.5%. For high system load, the measurements show that the client completion time for each application was an order of magnitude better using SWAP versus vanilla Linux. Despite the fact that the client was the highest priority process in the system, the video playback rate of MPEG play was only 0.72 frm/s when using vanilla Linux, which means that each video frame took on average 1389 ms to be processed and displayed. In the same situation, it took Netscape more than 11 seconds to download a single web page. In both cases the performance was unacceptable. The problem is that MPEG play and Netscape, as graphics-intensive applications, depend on the X Server to render the video frames and web pages. Since the X server was run at the same default priority as all the other VNC sessions in the system, this will effectively make all the clients depending on it run at low priority also.

On the other hand, Figure 5 shows the performance of both MPEG play and Netscape remained to be satisfactory even under very high system load when using SWAP . This further proves the effectiveness of the automatic dependency detection mechanism and dependency driven scheduler used by SWAP. The small difference between using SWAP for high and low system load can be explained by two factors. First, Linux still doesn't support features like a preemptive kernel which is important to real-time applications. Second, since access to resources such as memory and disk are not scheduled by the CPU scheduler, our SWAP CPU scheduling implementation does not solve performance degradation problems caused by accessing these resources.

5.4 Volano Chat Server

The Chat server workload consisted of VolanoMark 2.1.2 [7], an industry standard Java chat server benchmark configured in accordance with the rules of the Volano Report. VolanoMark creates a large number of threads and network connections, resulting in frequent scheduling and potentially many interprocess dependencies. It creates client connections in groups of 20 and measure how long it takes the clients to take turns broadcasting their messages to the group. It reports the average number of messages transferred by the server per second. For this experiment, all processes were run using the default SCHED_OTHER scheduling policy. We assumed that the chat clients are important and are therefore run as at a higher priority by running them with `nice -20` with all other applications run at the default priority. We measured the VolanoMark performance when there were no other applications running on the system to provide a baseline performance measure on vanilla Linux and SWAP. We then measured the VolanoMark performance with different levels of additional system load. The system load was generated using a simple CPU-bound application. To produce different system load levels, we ran different numbers of instances of the CPU-bound application. We ran VolanoMark using the dual-CPU server configuration. This provides a measures of the performance of vanilla Linux versus SWAP with a resource-intensive server application running on a loaded multiprocessor in the presence of process dependencies. VolanoMark was run using Sun's Java 2 Platform 1.4.0 for Linux which maps Java threads to Linux kernel threads in a one-to-one manner.

Figure 6 shows the performance of VolanoMark for different levels of system load. These results were obtained on the dual-CPU configuration of the server. The system load is equal to the number of additional CPU-bound applications running at the same time. For no additional system load, vanilla Linux averaged 4396 messages per second on VolanoMark test while SWAP averaged 4483. The measurements show that VolanoMark performs roughly the same for both vanilla Linux and SWAP, with the performance of SWAP slightly better. This can be explained by the fact that the Volano clients frequently call `sched_yield`, which allows the CPU scheduler to decide which client should run next. Because SWAP is aware of the dependency relationships among clients, SWAP can make a better decision than vanilla Linux regarding which client should run next.

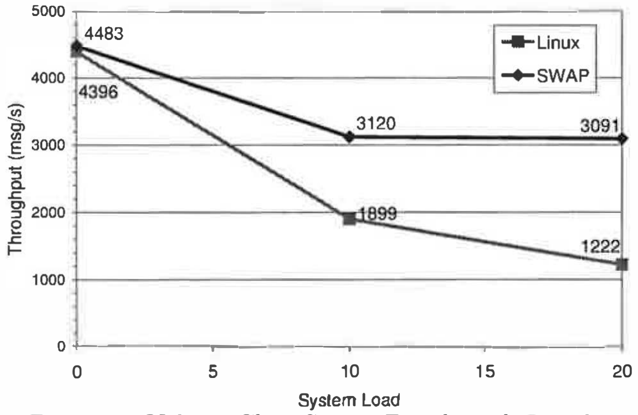


Figure 6: Volano Chat Server Benchmark Results

The performance of vanilla Linux and SWAP diverge significantly with additional system load. At a system load of 20, SWAP provides VolanoMark performance that is more than 2.5 times better than vanilla Linux. Although SWAP does perform much better than vanilla Linux, both systems show degradation in the performance of VolanoMark at higher system load. At a system load of 20, SWAP performance is about 70 percent of the maximum performance while vanilla Linux performance is less than 30 percent of the maximum performance. The degradation in performance on SWAP can be explained because of its reliance on the CPU scheduler to select among runnable and virtual runnable processes and the multiprocessor scheduling algorithm used in the Linux 2.4.18-14 kernel. For SWAP to deliver the best performance, high priority virtual runnable processes should always be scheduled before lower priority runnable processes. However, the Linux scheduler does not necessarily schedule in this manner on a multiprocessor. The Linux scheduler employs a separate run queue for each CPU and partitions processes among the run queues based on the number of runnable processes in each queue. It does not take into account the relative priority of processes in determining how to assign processes to run queues. As a result, for a two-CPU machine, the scheduler can end up assigning high priority processes to one CPU and lower priority processes to another. With SWAP, this can result in high priority virtual runnable processes competing for the same CPU even though lower priority processes are being run on the other CPU. As a result, some high priority virtual runnable processes end up having to wait in one CPU run queue when there are other lower priority CPU-bound applications which end up running on the other CPU.

Since Linux schedules processes independently, it does not account for the dependencies between client and server, resulting in high priority Volano clients not being able to run in the presence of other CPU-bound applications. Linux either delivers poor performance for these clients or places the burden on users to tune the performance of their applications by identifying process dependencies and explicitly raising the priority of all interdependent processes. SWAP instead relieves users of the burden of attempting to compensate for scheduler limitations. Our results show that SWAP automatically identifies the dynamic dependencies among processes and correctly accounts for them in scheduling to deliver better scheduling behavior and system performance.

6 Conclusions

Our experiences with SWAP and experimental results in the context of a general-purpose operating system demonstrate that SWAP is able to effectively and automatically detect process dependencies and accounts for these dependencies in scheduling. We show that SWAP effectively uses system call history to handle process dependencies such as those resulting from interprocess communication and synchronization mechanisms which have not been previously addressed. We also show that SWAP's confidence feedback model is effective in finding the fastest way to resolve process dependencies when multiple potential dependencies exist.

These characteristics of SWAP result in significant improvements in system performance when running applications with process dependencies. Our experimental results show that SWAP can provide more than an order of magnitude improvement in performance versus the popular Linux operating system when running microbenchmarks and real applications on a heavily loaded system. We show that SWAP can be integrated with existing scheduling mechanisms and operate effectively with schedulers that dynamically adjust priorities. Furthermore, our results show that SWAP achieves these benefits with very modest overhead and without any application modifications or any intervention by application developers or end users.

7 Acknowledgments

This work was supported in part by NSF grant EIA-0071954, an NSF CAREER Award, and an IBM SUR Award.

References

[1] Ted Baker. Stack-Based Scheduling of Real-Time Processes. *Real-Time Systems*, 3(1), March 1991.

[2] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, 1990.

[3] Raymond K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

[4] Dror G. Feitelson and Larry Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[5] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[6] Xavier Leroy. The LinuxThreads Library. Now a part of the glibc GNU C library.

[7] Volano LLC. Volanomark Benchmark. *http://www.volano.com/benchmarks.html*.

[8] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, first edition, 2000.

[9] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 35–48, Lancaster, U.K., 1993.

[10] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[11] R. Patel, K. Smith, and B. Liu. MPEG Video in Software: Representation, Transmission, and Playback. In *Proc. High-Speed Networking and Multimedia Computing*, San Jose, California, February 8-10 1994.

[12] Glenn E Reeves. What Really Happened on Mars Rover Pathfinder. *The Risks Digest*, 19, 1997.

[13] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[14] Manfred Ruschitzka and Robert S. Fabry. A Unifying Approach to Scheduling. *Communications of the ACM*, 20(7):469–477, July 1977.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[16] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. *Lecture Notes in Computer Science*, 1459:231–257, 1998.

[17] S. Sommer. Removing Priority Inversion from an Operating System. In *Proceedings of Nineteenth Australasian Computer Science*, 1996.

[18] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 145–158, Berkeley, CA, February 22–25 1999. Usenix Association.

[19] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.

Contract-Based Load Management in Federated Distributed Systems*

Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker
MIT Computer Science and Artificial Intelligence Lab
`http://nms.lcs.mit.edu/projects/medusa/`

Abstract

This paper focuses on load management in loosely-coupled federated distributed systems. We present a distributed mechanism for moving load between autonomous participants using bilateral contracts that are negotiated offline and that set bounded prices for moving load. We show that our mechanism has good incentive properties, efficiently redistributes excess load, and has a low overhead in practice.

Our load management mechanism is especially well-suited for distributed stream-processing applications, an emerging class of data-intensive applications that employ a "continuous query processing" model. In this model, streams of data are processed and composed continuously as they arrive rather than after they are indexed and stored. We have implemented the mechanism in the *Medusa* distributed stream processing system, and we demonstrate its properties using simulations and experiments.

1 Introduction

Many distributed systems are composed of loosely coupled autonomous nodes spread across different administrative domains. Examples of such federated systems include Web services, cross-company workflows where the end-to-end services require processing by different organizations [3, 21], and peer-to-peer systems [8, 23, 30, 45]. Other examples are computational grids composed of computers situated in different domains [4, 16, 44], overlay-based computing platforms such as Planetlab [35], and data-intensive stream processing systems [1, 2, 5, 6, 7] that can be distributed across different domains to provide data management services for data streams.

Federated operation offers organizations the opportunity to pool their resources together for common benefit. Participants can compose the services they provide into more complete end-to-end services. Organizations can also cope with load spikes without individually having to maintain and administer the computing, network, and storage resources required for peak operation.

Autonomous participants, however, do not collaborate for the benefit of the whole system, but rather aim to maximize their own benefit. A natural way to architect a federated system is thus as a *computational economy*, where participants provide resources and perform computing for each other in exchange for payment.[1]

When autonomous participants are also real economic entities, additional constraints come into play. The popularity of bilateral agreements between Internet Service Providers (ISPs) demonstrates that participants value and even require privacy in their interactions with each other. They also practice price and service discrimination [24], where they offer different qualities of service and different prices to different partners. For this purpose, ISPs establish bilateral Service Level Agreements, where they define confidential details of the *custom* SLA and prices that one partner offers another.

In this paper, we present a distributed mechanism for managing load in a federated system. Our mechanism is inspired on the manner in which ISPs collaborate. Unlike other computational economies that implement global markets to set resource prices at runtime, our mechanism is based on *private pairwise contracts* negotiated offline between participants. Contracts set tightly *bounded prices* for migrating each unit of load between two participants and specify the set of tasks that each is willing to execute on behalf of the other. We envision that contracts will be extended to contain additional clauses further customizing the offered services (e.g., performance and availability guarantees). In contrast to previous proposals, our mechanism (1) provides privacy to all participants regarding the details of their interactions with others, (2) facilitates service customization and price discrimination, (3) provides a simple and lightweight runtime load management using price pre-negotiation, and (4) has good system-wide load balance properties.

With this *bounded-price mechanism*, runtime load transfers occur only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. The load transfer mechanism is simple: a participant moves load to another if the local processing cost is larger than the payment it would have to make to

*This material is based upon work supported by the National Science Foundation under Grant No. 0205445. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[1]Non-payment models, such as bartering, are possible too. See Section 2 for details.

another participant for processing the same load (plus the migration cost).

Our work is applicable to a variety of federated systems, and is especially motivated by *distributed stream processing applications*. In these applications, data streams are continuously pushed to servers, where they undergo significant amounts of processing including filtering, aggregation, and correlation. Examples of applications where this "push" model for data processing is appropriate include financial services (*e.g.*, price feeds), medical applications (*e.g.*, sensors attached to patients), infrastructure monitoring (*e.g.*, computer networks, car traffic), and military applications (*e.g.*, target detection).

Stream processing applications are well-suited to the computational economy provided by a federated system. Data sources are often distributed and belong to different organizations. Data streams can be composed in different ways to create various services. Stream processing applications also operate on large volumes of data, with rates varying with time and often exceeding tens of thousands of messages per second. Supporting these applications thus requires dynamic load management. Finally, because the bulk of the processing required by applications can be expressed with standard well-defined operators, load movements between autonomous participants does not require full-blown process migration.

We have designed and implemented the bounded-price mechanism in *Medusa*, a federated distributed stream-processing system. Using analysis and simulations, we show that the mechanism provides enough incentives for selfish participants to handle each other's excess load, improving the system's load distribution. We also show that the mechanism efficiently distributes excess load when the aggregate load both underloads and overloads total system capacity and that it reacts well to sudden shifts in load. We show that it is sufficient for contracts to specify a small price-range in order for the mechanism to produce acceptable allocations where (1) either *no* participant operates above its capacity, or (2) if the system as a whole is overloaded, then *all* participants operate above their capacity. We further show that the mechanism works well even when participants establish heterogeneous contracts at different unit prices with each other.

We discuss related work in the next section. Section 3 presents the bounded-price load management mechanism and Section 4 describes its implementation in Medusa. We present several simulation and experimental results in Section 5 and conclude in Section 6.

2 Related Work

Cooperative load sharing in distributed systems has been widely studied (see, *e.g.*, [10, 19, 22, 25, 41]). Approaches most similar to ours produce optimal or near-optimal allocations using *gradient-descent*, where nodes exchange load among themselves producing successively less costly allocations. In contrast to these approaches, we focus on environments where participants are directed by self-interest and not by the desire to produce a system-wide optimal allocation.

As recent applications frequently involve independently administered entities, more efforts have started to consider participant selfishness. In mechanism design (MD) [20, 33], agents reveal their costs to a central entity that computes the optimal allocation and a vector of compensating payments. Agents seek to maximize their utility computed as the difference between payment received and processing costs incurred. Allocation and payment algorithms are designed to optimize agents utility when the latter reveal their true costs.

In contrast to pure mechanism design, algorithmic mechanism design (AMD) [29, 32] additionally considers the computational complexity of mechanism implementations. Distributed algorithmic mechanism design (DAMD) [12, 14] focuses on distributed implementations of mechanisms, since in practice a central optimizer may not be implementable. Previous work on DAMD schemes includes BGP-based routing [12] and cost-sharing of multicast trees [13]. These schemes assume that participants correctly execute payment computations. In contrast, our load management mechanism is an example of a DAMD scheme that does not make any such assumption because it is based on bilateral contracts.

Researchers have also proposed the use of economic principles and market models for developing complex distributed systems [27]. Computational economies have been developed in application areas such as distributed databases [43], concurrent applications [46], and grid computing [4, 16, 44]. Most approaches use pricing [4, 9, 15, 16, 39, 43, 46]: resource consumers have different price to performance preferences and are allocated a budget. Resource providers hold auctions to determine the price and allocation of their resources. Alternatively, resource providers bid for tasks [39, 43], or adjust their prices iteratively until demand matches supply [15].

These approaches to computational economies require participants to hold and participate in auctions for every load movement, thus inducing a large overhead. Variable load may also make prices vary greatly and lead to frequent re-allocations [15]. If the cost of processing clusters of tasks is different from the cumulative cost of independent tasks, auctions become combinatorial [31, 34][2], complicating the allocation problem. If auctions are held by overloaded agents, underloaded agents have the choice to participate in one or many auctions simultaneously, leading to complex market clearance and exchange mechanisms [29]. We avoid these complexities by bounding

[2] In a combinatorial auction, multiple items are sold concurrently. For each bidder, each subset of these items represents a different value.

the variability of runtime resource prices and serializing communications between partners. In contrast to our approach, computational economies also make it significantly more difficult for participants to offer different prices and different service levels to different partners.

As an alternative to pricing, recent approaches propose to base computational economies on *bartering*. SHARP [17] is an infrastructure that enables peers to securely exchange tickets that provide access to resources. SHARP does not address the policies that define how the resources should be exchanged. Chun *et al.* [8] propose a computational economy based on SHARP. In their system, peers discover required resources at runtime and trade resource tickets. A ticket is a soft claim on resources and can be rejected resulting in zero value for the holder. In contrast, our pairwise agreements do not specify any resource amounts and peers pay each other only for the resources they actually use.

Service level agreements (SLAs) are widely used for Web services and electronic commerce [3, 21, 37]. The contract model we propose fits well with these SLA infrastructures.

In P2P systems, peers offer their resources to each other for free. Schemes to promote collaboration use reputation [23], accounting [45], auditing [30], or strategyproof computing [28] to eliminate "free-riders" who use resources without offering any in return. In contrast, we develop a mechanism for participants that require tight control over their collaborations and do not offer their resources for free.

3 The Bounded-Price Mechanism

In this section, we define the load management problem in federated distributed systems, present the bounded-price mechanism and discuss its properties.

3.1 Problem Statement

We are given a system comprised of a set S of autonomous *participants* each with computing, network, and storage resources, and a time varying set K of heterogeneous *tasks* that impose a load on participants' resources. Each task is considered to originate at a participant where it is submitted by a client. Since we only examine interactions between participants, we use the terms *participant* and *node* interchangeably. Tasks can be aggregated into larger tasks or split into subtasks. If the load imposed by a task increases, the increase can thus be treated as the arrival of a new task. Similarly, a load decrease can be considered as the termination of a task. We discuss tasks further in Section 4.

For each participant, the load imposed on its resources represents a cost. We define a real-valued *cost function* of each participant i as:

$$\forall i \in S, \forall \text{taskset}_i \subseteq K, \quad D_i : \text{taskset}_i \longrightarrow \mathcal{R} \qquad (1)$$

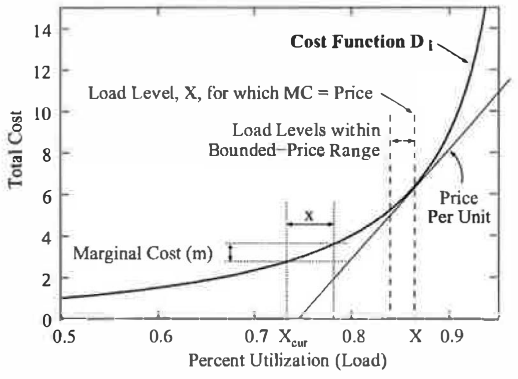


Figure 1: Prices and processing costs.

where taskset_i is the subset of tasks in K running at i. This cost depends on the load imposed by the tasks. Each participant monitors its own load and computes its processing cost. There are an unlimited number of possible cost functions and each participant may have a different one. We assume, however, that this cost is a *monotonic* and *convex* function. Indeed, for many applications that process messages (e.g., streams of tuples), an important cost metric is the per message processing delay. For most scheduling disciplines this cost is an increasing and convex function, reflecting the increased difficulty in offering low delay service at higher load. Figure 1 illustrates such cost function for a single resource. We revisit Figure 1 further throughout the section.

Participants are selfish and aim to maximize their utility, computed as the difference between the processing cost, $D_i(\text{taskset}_i)$, and the payment they receive for that processing. When a task originates at a participant, it has a constant per-unit load value to that participant (this value could be, for instance, the price paid by the participant's clients for the processing). When a task comes from another participant, the payment made by that participant defines the task's value.

Each participant has a maximum load level corresponding to a maximum cost, above which the participant considers itself overloaded. The goal of a load management mechanism is to ensure that no participant is overloaded, when spare capacity exists. If the whole system is overloaded, the goal is to use as much of the available capacity as possible. We seek a mechanism that produces an *acceptable allocation*:

Definition: An **acceptable allocation** is a task distribution where (1) *no* participant is above its capacity threshold, *or* (2) *all* participants are at or above their capacity thresholds if the total offered load exceeds the sum of the capacity thresholds.

Because the set of tasks changes with time, the allocation problem is an online optimization. Since the system is a federation of loosely coupled participants, no single entity can play the role of a central optimizer and the implementation of the mechanism must be distributed. We further examine the mechanism design aspects of our approach in Section 3.3.

In our scheme, load movements are based on *marginal costs*, $\text{MC}_i : (u, \text{taskset}_i) \rightarrow \mathcal{R}$ defined as the incremental cost for node i of running task u given its current taskset_i. Figure 1 shows the marginal cost m caused by adding load x, when the current load is X_{cur}. Assuming the set of tasks in taskset_i imposes a total load X_{cur} and u imposes load x, then $\text{MC}(u, \text{taskset}_i) = m$. If x is one unit of load, we call m the *unit marginal cost*.

3.2 Model and Algorithms

We propose a mechanism to achieve acceptable allocations based on bilateral contracts: participants establish contracts with each other by negotiating offline a set of tightly bounded prices for each unit of load they will move in each direction.

Definition: A **contract** $\mathcal{C}_{i,j}$ between participants i and j defines a price range: $[\text{min\_price}(\mathcal{C}_{i,j}), \text{max\_price}(\mathcal{C}_{i,j})]$, that constrains the runtime price paid by participant i for each unit of load given to j.

Participants must mutually agree on what one unit of processing, bandwidth, and storage represent. Different pairs of participants may have contracts specifying different unit prices. There is at most one contract for each pair of participants and each direction. Participants may periodically renegotiate, establish, or terminate contracts offline. We assume that the set of nodes and contracts form a connected graph. The set of a participant's contracts is called its contractset. We use C to denote the maximum number of contracts that any participant has.

At runtime, participants that have a contract with each other may perform *load transfers*. Based on their load levels, they agree on a definite unit price, $\text{price}(\mathcal{C}_{i,j})$, within the contracted price-range, and on a set of tasks, the moveset, that will be transferred. The participant offering load also pays its partner a sum of $\text{price}(\mathcal{C}_{i,j}) * \text{load}(\text{moveset})$.

3.2.1 Fixed-Price Contracts

We first present the *fixed-price mechanism*, where $\text{min\_price}(\mathcal{C}_{i,j}) = \text{max\_price}(\mathcal{C}_{i,j}) = \text{FixedPrice}(\mathcal{C}_{i,j})$. With fixed-price contracts, if the marginal cost per unit of load of a task is higher than the price in a contract, then processing that task locally is more expensive than paying the partner for the processing. Conversely, when a task's marginal cost per unit of load is below the price specified in a contract, then accepting that task results in a greater payment than cost increase.

Given a set of contracts, we propose a load management protocol, where each participant concurrently runs one algorithm for shedding excess load (Figure 2) and one for taking on new load (Figure 3).

```
PROCEDURE OFFER_LOAD:
repeat forever:
    sort(contractset on price(contractset_j) ascending)
    foreach contract $C_j \in$ contractset:
        offerset $\leftarrow \emptyset$
        foreach task $u \in$ taskset
            total_load $\leftarrow$ taskset $-$ offerset $- \{u\}$
            if MC($u$, total_load) $>$ load($u$) $*$ price($\mathcal{C}_j$)
                offerset $\leftarrow$ offerset $\cup \{u\}$
        if offerset $\neq \emptyset$
            offer $\leftarrow$ (price($\mathcal{C}_j$), offerset)
            (resp, acceptset) $\leftarrow$ send_offer($j$, offer)
            if resp = accept and acceptset $\neq \emptyset$
                transfer($j$, price($\mathcal{C}_j$), acceptset)
                break foreach contract
    wait $\Omega$ time units
```

Figure 2: Algorithm for shedding excess load.

The basic idea in shedding excess load is for an overloaded participant to select a maximal set of tasks from its taskset_i that cost more to process locally than they would cost if processed by one of its partners and offer them to that partner. Participants can use various algorithms and policies for selecting these tasks. We present a general algorithm in Figure 2. If the partner accepts even a subset of the offered tasks, the accepted tasks are transferred. An overloaded participant could consider its contracts in any order. One approach is to exercise the lower-priced contracts first with the hope of paying less and moving more tasks. In this paper, we ignore the task migration costs. These costs should, however, be considered before an offer is sent by imposing a minimum threshold between the difference in the local and remote processing costs.

Procedure OFFER_LOAD waits between load transfers to let local load level estimations (e.g., exponentially weighted moving averages) catch-up with the new average load level. If no transfer is possible, a participant retries to shed load periodically. Alternatively, the participant may ask its partners to notify it when their loads decrease sufficiently to accept new tasks.

In procedure ACCEPT_LOAD (Figure 3), each participant continuously accumulates load offers and periodically accepts subsets of offered tasks, examining the higher unit-price offers first. Since accepting an offer results in a load movement (because offers are sent to one partner at the time), the participant keeps track of all accepted tasks in the potentialset and responds to both accepted and rejected offers. Participants that accept a load offer cannot cancel transfers and move tasks back when load conditions change. They can, however, use their own contracts to move load further or to move it back.

There are several advantages in serializing communication between participants rather than having them offer their load simultaneously to all their partners. The

```
PROCEDURE ACCEPT_LOAD:
repeat forever:
    offers ← ∅
    for Ω time units or while (movement = true)
        for each new offer received, new_offer:
            offers ← offers ∪ {new_offer}
    sort(offers on price(offers_i) descending)
    potentialset ← ∅
    foreach offer $o_i$ ∈ offers
        acceptset ← ∅
        foreach task $u$ ∈ offerset($o_i$)
            total_load ← taskset ∪ potentialset ∪ acceptset
            if MC($u$, total_load) < load($u$) * price($o_i$)
                acceptset ← acceptset ∪ {$u$}
        if acceptset ≠ ∅
            potentialset ← potentialset ∪ acceptset
            resp ← (accept, acceptset)
            movement ← true
        else resp ← (reject, ∅)
        respond($o_i$, resp)
```

Figure 3: Algorithm for taking additional load.

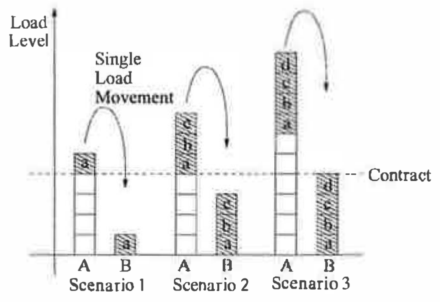


Figure 4: Three load movement scenarios for two partners.

communication overhead is lower but most importantly, the approach prevents possible overbooking as partners always receive the load they accept. This in turn allows participants to accept many offers at once. The only drawback is a slightly longer worst-case convergence time.

Figure 4 illustrates three load movement scenarios. If participant A has one or more tasks for which its marginal cost per unit of load exceeds the price in its contract with B, these tasks are moved in a single transfer (scenarios 1 and 2). Only those tasks, however, for which the marginal cost per unit of load at B does not exceed the price in the contract are transferred (scenario 3).

3.2.2 Setting Up Fixed-Price Contracts

With an approach based on fixed prices, the only tunable parameters are the unit prices set in contracts. When a participant negotiates a contract to shed load, it must first determine its maximum desired load level X, and the corresponding marginal cost per unit of load. This marginal cost is also the maximum unit price that the participant should accept for a contract. For any higher price, the participant risks being overloaded and yet unable to shed load. Any price below that maximum is acceptable. Figure 1 illustrates, for a single resource and a strictly convex function, how a load level X maps to a unit price. In general, this price is the gradient of the cost function evaluated at X.

When a participant negotiates a contract to accept load, the same maximum price rule applies since the participant will never be able to accept load once it is overloaded itself. A participant, however, should not accept a price that is too low because such price would prevent the participant from accepting new load even though it might still have spare capacity. The participant should rather estimate its expected load level, select a desired load level between the expected and maximum levels and negotiate the corresponding contract price.

Participants may be unwilling to move certain tasks to some partners due to the sensitive nature of the data processed or because the tasks themselves are valuable intellectual property. For this purpose, contracts can also specify the set of tasks (or types of tasks) that may be moved, constraining the runtime task selection. In offline agreements, participants may also prevent their partners from moving their operators further thus constraining the partner's task selections.

To ensure that a partner taking over a task provides enough resources for it, contracts may also specify a minimum per-message processing delay (or other performance metric). A partner must meet these constraints or pay a monetary penalty. Such constraints are commonplace in SLAs used for Web services, inter-company workflows or streaming services [18]. Infrastructures exist to enforce them through automatic verification [3, 21, 37, 38]. In the rest of this paper, we assume that such infrastructure exists and that monetary penalties are high enough to discourage any contract breaches. To avoid breaching contracts when load increases, participants may prioritize tasks already running over newly arriving tasks.

3.2.3 Extending the Price Range

Fixed-price contracts do not always produce acceptable allocations. For instance, load cannot propagate through a chain of identical contracts. A lightly loaded node in the middle of a chain accepts new tasks as long as its marginal cost is *strictly below* the contract price. The node eventually reaches maximum capacity (as defined by the contract prices) and refuses additional load. It does not offer load to partners that might have spare capacity, though, because its unit marginal cost is still lower than any of its contract prices. Hence, if all contracts are identical, a task can only propagate one hop away from its origin.

To achieve acceptable allocations for all configurations, participants thus need to specify a *small range of prices*, [FixedPrice − Δ; FixedPrice], in their contracts. Such price range allows a participant to forward load from an overloaded partner to a more lightly loaded one by accepting tasks at a higher price and offering them at a lower price. When a contract specifies a small price-range, for each load transfer, partners must dynamically negotiate the final unit price within the range. Since a fixed unit price equals the gradient (or derivative) of the cost curve at some load level, a price range converts into a load level interval as illustrated in Figure 1. The price range is the difference in the gradients of the cost curve at interval boundaries.

We now derive the minimal contract price-range that ensures convergence to acceptable allocations. We analyze a network of homogeneous nodes with identical contracts. We explore heterogeneous contracts through simulations in Section 5. For clarity of exposition, we also assume in the analysis that all tasks are identical to the smallest migratable task, u and *impose the same load*. We use ku to denote a set of k tasks.

We define δ_k as the decrease in unit marginal cost due to removing k tasks from a node's taskset:

$$\delta_k(\text{taskset}) = \frac{\text{MC}(u, \text{taskset} - u) - \text{MC}(u, \text{taskset} - (k+1)u)}{\text{load}(u)} \tag{2}$$

δ_k is thus approximately the difference in the cost function gradient evaluated at the load level including and excluding the k tasks.

Given a contract with price, FixedPrice, we define taskset^F as the maximum set of tasks that a node can handle before its per-unit-load marginal cost exceeds FixedPrice and triggers a load movement. I.e., taskset^F satisfies: $\text{MC}(u, \text{taskset}^\text{F} - u) \leq \text{load(u)} * \text{FixedPrice}$ and $\text{MC}(u, \text{taskset}^\text{F}) > \text{load(u)} * \text{FixedPrice}$.

With all contracts in the system specifying the same price range, $[\text{FixedPrice} - \Delta, \text{FixedPrice}]$ such that $\Delta = \delta_1(\text{taskset}^\text{F})$, any task can now travel two hops. A lightly loaded node accepts tasks at FixedPrice until its load reaches that of taskset^F. The node then alternates between offering one task u at price $\text{FixedPrice} - \delta_1(\text{taskset}^\text{F})$ and accepting one task at FixedPrice. This scenario is shown in Figure 5. Similarly, for load to travel through a chain of $M + 1$ nodes (or M transfers) the price range must be at least $\delta_{M-1}(\text{taskset}^\text{F})$. The jth node in such a chain alternates between accepting a task at price $\text{FixedPrice} - \delta_{j-1}(\text{taskset}^\text{F})$ and offering it at price $\text{FixedPrice} - \delta_j(\text{taskset}^\text{F})$.

A larger price range speeds-up load movements through a chain because more tasks can be moved at each step. With a larger price range, however, nodes unit marginal costs are more likely to fall within the dynamic range requiring a price negotiation. A larger range thus increases runtime overhead, price volatility and the number of re-allocations caused by small load variations. Our goal is therefore to keep the range as small as possible and extend it only enough to ensure convergence to acceptable allocations.

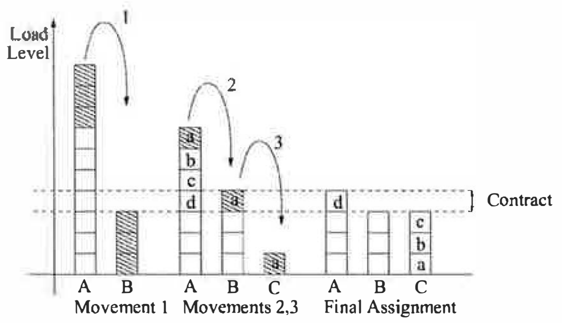


Figure 5: Load movements between three nodes using a small price-range.

For underloaded systems to always converge to acceptable allocations, tasks must be able to travel as far as the diameter M of the network of contracts. The minimal price range should then be $\delta_{M-1}(\text{taskset}^\text{F})$.

Lemma 1 *In a network of homogeneous nodes, tasks, and contracts, to ensure convergence to acceptable allocations in underloaded systems, the unit price range in contracts must be at least:* $[\text{FixedPrice} - \delta_{M-1}(\text{taskset}^\text{F}), \text{FixedPrice}]$, *where M is the diameter of the network of contracts and* taskset^F *is the set of tasks that satisfies* $\text{MC}(u, \text{taskset}^\text{F} - u) \leq \text{load(u)} * \text{FixedPrice}$ *and* $\text{MC}(u, \text{taskset}^\text{F}) > \text{load(u)} * \text{FixedPrice}$.

When the system is overloaded, a price range does not lead to an acceptable-allocation (where $\forall i \in S : D_i(\text{taskset}_\text{i}) \geq D_i(\text{taskset}_\text{i}^\text{F})$). In the final allocation, some participants may have a marginal cost as low as $\text{FixedPrice} - \delta_M(\text{taskset}_\text{i}^\text{F})$ (wider ranges do not improve this bound). For overloaded systems, price-range contracts therefore achieve *nearly acceptable allocations* defined as:

Definition: A **nearly acceptable allocation** satisfies $\forall i \in S : D_i(\text{taskset}_\text{i}) > D_i(\text{taskset}_\text{i}^\text{F} - Mu)$

Price ranges modify the load management protocol as follows. Initial load offers are made at the lowest price. If no task can be accepted, the partner proposes a higher price. Upon receiving such a counter-offer, if the new price is still its best alternative, a participant recomputes the offerset. If the set is empty, it suggests a new price in turn. Negotiation continues until both parties agree on a price or no movement is possible. Other negotiation schemes are possible.

3.3 Properties

The goal of mechanism design [33] is to implement an optimal system-wide solution to a decentralized optimization problem, where each agent holds an input parameter to the problem and prefers certain solutions over others. In our case, agents are participants and optimization parameters are participants' cost functions and original sets of tasks. The system-wide goal is to achieve an acceptable allocation while each participant tries to optimize its utility. Our mechanism is *indirect*: participants reveal their costs and tasks indirectly by offering and accepting tasks rather than announcing their costs directly to a central optimizer or to other agents.

A mechanism defines the set of strategies S available to agents and an outcome rule, $g : S^N \rightarrow O$, that maps all possible combinations of strategies adopted by agents to an outcome O. With fixed-price contracts, the runtime strategy-space of participants is reduced to only three possible strategies: (1) accept load at the pre-negotiated price, (2) offer load at the pre-negotiated price, or (3) do neither.[3] The desired outcome is an acceptable allocation.

Similarly to the definition used in Sandholm *et. al.* [40], our mechanism is *individual rational* (i.e., a participant may not decrease its utility by participating) on a per load movement basis. Each agent increases its utility by accepting load when the price exceeds the per-unit-load marginal cost (because in that case the increase in payment, p_i, exceeds the increase in cost, D_i) and offer load in the opposite situation. For two agents and one contract this strategy is also *dominant* because compared to any other strategy, it optimizes an agent's utility independently of what the other agent does.

For multiple participants and contracts, the strategy space is richer. Participants may try to optimize their utility by accepting too much load with the hope of passing it on at a lower price. Assuming, however, that participants are highly *risk-averse*: they are unwilling to take on load unless they can process it at a lower cost themselves because they risk paying monetary penalties, the strategy of offering and accepting load *only* when marginal costs are strictly higher or lower than prices respectively, is an optimal strategy. This strategy is not dominant, though, because it is technically possible that a participant has a partner that always accepts load at a low price. In specific situations, the order in which participants contact each other may also change their utility due to simultaneous moves by other participants.

These properties also hold for price-range contracts, when participants' marginal costs are far from range boundaries. Within a range, participants negotiate, thus revealing their costs more directly. Reaching an agreement is individual-rational since moving load at a price between participants' marginal costs increases both their utilities. Partners thus agree on a price within the range, when possible.

Our mechanism is also a distributed algorithmic mechanism (DAM) [12, 14] since the implementation is (1) distributed: there is no central optimizer, and (2) algorithmic: the computation and communication complexities are both polynomial time, as we show below. Because our mechanism is indirect, it differs from previous DAMD approaches [12, 14] that focus on implementing the same payment computation as would a central optimizer but in an algorithmic and distributed fashion. An important assumption made in these implementations is that agents are either separate from the entities computing the payment functions [14] or that they compute the payments honestly [12]. Our mechanism does not need to make any such assumption.

Because each load transfer takes place only if the marginal cost of the node offering load is strictly greater than that of the node accepting the load, successive allocations strictly decrease the sum of all costs. Under constant load, movements thus always eventually stop. If all participants could talk to each other, the final allocation would always be acceptable and *Pareto-optimal*: i.e., no agent could improve its utility without another agent decreasing its own utility. In our mechanism, however, participants only exchange load with their direct partners and this property does not hold. Instead, for a given load, the bounded-price mechanism limits the maximum difference in load levels that can exist between participants once the system converges to a final allocation. If a node has at least one task for which the unit marginal cost is greater than the upper-bound price of any of the node's contracts, then all its partners must have a load level such that an additional task would have an average unit marginal cost greater than the upper-bound price in their contract with the overloaded node. If a partner had a lower marginal cost, it would accept its partner's excess task at the contracted price. This property and the computation of the minimal price ranges yield the following theorem:

Theorem 2 *If nodes, contracts and tasks are homogeneous, and contracts are set according to Lemma 1, the final allocation is an acceptable allocation for underloaded systems and a nearly acceptable allocation for overloaded systems.*

In Section 5, we analyze heterogeneous settings using simulations and find that in practice, nearly acceptable allocations are also reached in such configurations.

Another property of the fixed-price mechanism is its fast convergence and low communication overhead. Task selection is the most complex operation and is performed once for each load offer and once for each response. Therefore, in a system with αN overloaded participants, under constant load, in the best case, all excess tasks require a single offer and are moved in parallel, for a convergence time of $O(1)$. In the worst case, the overloaded participants form a chain with only the last participant in the chain having spare capacity. In this configuration, participants must shed load one at the time, through the most ex-

[3]We exclude the task selection problem from the strategy space.

pensive of their C contracts, for a worst case convergence time of $O(NC)$. If nodes use notifications when they fail to shed load, the worst-case is reduced to $O(N+C)$. For auctions, the worst-case convergence time is $O(N)$ in this configuration. To summarize:

Lemma 3 *For N nodes with at most C contracts each, the fixed-price mechanism has a convergence time of $O(1)$ in the best case and $O(N+C)$ in the worst case if notifications are used.*

With our mechanism, most load movements require as few as three messages: an offer, a response, and a load movement. The best-case communication complexity is thus $O(N)$. In contrast, an approach based on auctions has a best-case communication overhead of $O(NC)$ if the auction is limited to C partners and $O(N^2)$ if not. If the whole system is overloaded and participants must move load through a chain of most-expensive contracts, the worst-case complexity may be as high as $O(N^2C)$, mostly because each participant periodically tries to shed load. If, however, the notifications described above are used, the worst-case communication overhead is only $O(NC)$. The same worst-case communication overhead applies to auctions. In summary:

Lemma 4 *To converge, the fixed-price contracts mechanism imposes a best-case communication overhead of $O(N)$ and a worst-case overhead of $O(NC)$ if notifications are used.*

Compared to auctions, our scheme significantly reduces the communication overhead, for a slight increase in worst-case convergence time. Simulation results (Section 5) show that the convergence time is short in practice.

When contracts specify a small price-range, most load movements take place outside of the range and the mechanism preserves the properties above. If, however, the optimal load allocation falls exactly within the small price-range, final convergence steps require more communication and may take longer to achieve, but at that point, the allocations are already acceptable. When load is forwarded through a chain, the complexity grows with the length of the chain and the amount of load that needs to be forwarded through that chain. In practice, though, long chains of overloaded nodes are a pathological, thus rare, configuration.

4 System Implementation

In this section, we describe the implementation of the bounded-price mechanism in the Medusa distributed stream-processing system.

4.1 Streams and Operators

In stream-processing applications, *data streams* produced by sensors or other data sources are correlated and aggregated by *operators* to produce outputs of interest. A data stream is a continuous sequence of attribute-value tuples that conform to some pre-defined schema. Operators are functions that transform one or more input streams into one or more output streams. A loop-free, directed graph of operators is called a *continuous query*, because it continuously processes tuples pushed on its input streams.

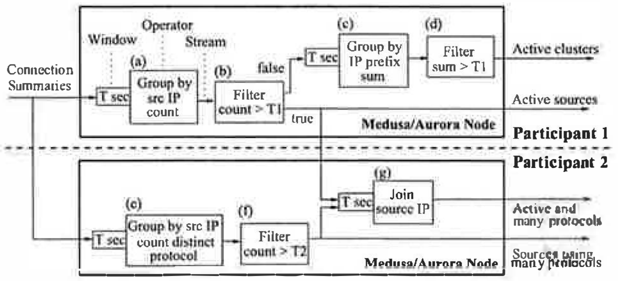


Figure 6: Example of a distributed Medusa query.

Stream processing applications are naturally distributed. Many applications including traffic management and financial feed analysis process data from either geographically distributed sources or different autonomous organizations. Medusa uses Aurora [1] as its query processor. Medusa takes Aurora queries and arranges for them to be distributed across nodes and participants, routing tuples and results as appropriate.

Figure 6 shows an example of a Medusa/Aurora query. The query, inspired from Snort [36] and Autofocus [11], is a simple network intrusion detection query. Tuples on input streams summarize one network connection each: source and destination IPs, time, protocol used, etc. The query identifies sources that are either active (operators `a` and `b`) or used abnormally large numbers of protocols within a short time period (operators `e` and `f`) or both (operator `g`). The query also identifies clusters of active sources (operators `c` and `d`). To count the number of connections or protocols the query applies *windowed aggregate* operators (`a`, `c`, and `e`): these operators buffer connection information for a time period T, group the information by source IP and apply the desired aggregate function. Aggregate values are then *filtered* to identify the desired type of connections. Finally, operator `g` *joins* active sources with those using many protocols to identify sources that belong in both categories.

The phrases in italics in the previous paragraph correspond to well-defined operators. While the system allows user-defined operators, our experience with a few applications suggests that developers will implement most application logic with built-in operators. In addition to simplifying application construction and providing query optimization opportunities, using standard operators facilitates Medusa's task movements between participants.

Medusa participants use *remote definitions* to move

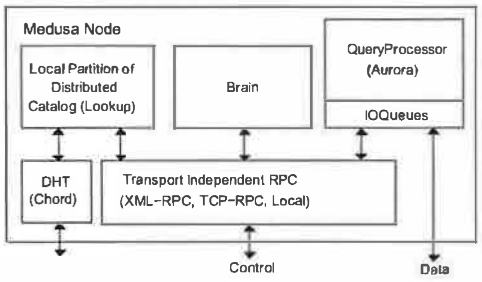

Medusa Node
Local Partition of Distributed Catalog (Lookup)
Brain
QueryProcessor (Aurora)
IOQueues
DHT (Chord)
Transport Independent RPC (XML-RPC, TCP-RPC, Local)
Control
Data

Figure 7: Medusa software structure.

tasks with relatively low overhead compared to full-blown process migration. Remote definitions specify how operators on different nodes map on to each other. At runtime, when a path of operators in the boxes-and-arrows diagram needs to be moved to another node, all that's required is for the operators to be instantiated remotely and for the incoming streams to be diverted to the appropriately named inputs on the new node. Our current prototype does not move operator state. The new instance re-starts the computation from an empty state. We plan to explore moving operator state in future work.[4]

For stream-processing, the algorithm for selecting tasks to offload to another participant must take into account the data flow between operators. It is preferable to cluster and move together operators that are linked with a high-rate stream or even simply belong to the same continuous query. In this paper we investigate the general federated load management problem and do not take advantage of possible relations between tasks to optimize their placement. In stream processing it is also often possible to partition input streams [41] and by doing so handle the load increase of a query network as a new query network. We make this assumption in this paper.

4.2 Medusa Software Architecture

Figure 7 shows the Medusa software structure. Each Medusa node runs one instance of this software. There are two components in addition to the Aurora query processor. The first component, called `Lookup`, is a client of an inter-node distributed catalog (implemented using a distributed hash table [42]) that holds information on streams, schemas and queries running in the system. The `Brain` component monitors local load conditions by periodically asking the `QueryProcessor` for the average input and output rates measured by the `IOQueues` (which serve to send and receive tuples to and from clients or other Medusa nodes) as well as for rough estimates of the local CPU utilization of the various operators. `Brain` uses this load information as input to the bounded-price mechanism that manages load.

[4]The semantics of many stream processing applications are such that occasional tuple drops are acceptable [2].

min # of contracts	1	3	5	7	9	10
max # of contracts	11	13	14	15	17	18
avg diameter	19	8	6	5	4	4

Table 1: Max number of contracts per node and network diameter for increasing min numbers of contracts.

5 Evaluation

In previous sections, we showed some properties of our mechanism and computed the best- and worst-case complexities. In this section, we complete the evaluation using simulations and experiments. We first examine the convergence to acceptable allocations in a network of heterogeneous nodes. We simulate heterogeneity by setting contracts at randomly selected prices. Second, we study the average convergence speed in large and randomly created topologies. Third, we examine how our approach to load management reacts to load variations. Finally, we examine how Medusa performs on a real application by running the network intrusion detection query, presented in Section 4, on logs of network connections.

We use the CSIM discrete event simulator [26] to study a 995-node Medusa network. We simulate various random topologies, increasing the minimum number of contracts per node, which has the effect of reducing the diameter of the contract network. To create a contract network, each node first establishes a bilateral contract with an already connected node, forming a tree. Nodes that still have too-few contracts then randomly select additional partners. With this algorithm, the difference between the numbers of contracts that nodes have is small, as shown in Table 1.

Each node runs a set of independent and identical operators that process tuples at a cost of 50 μs/tuple. We set the input rate on each stream to 500 tuples/s (or 500 KBps). Assuming that each node has one 100 Mbps output link, each operator uses 4% of the bandwidth and 2.5% of CPU. We select these values to model a reasonable minimum migratable unit. In practice, tasks are not uniform and the amount of resources each task consumes bounds how close a participant's marginal cost can get to a contract price without crossing it. When we examine convergence properties, we measure overload and available capacity in terms of the number of tasks that can be offered or accepted, rather than exact costs or load. All nodes use the same convex cost function: the total number of in-flight tuples (tuples being processed and tuples awaiting processing in queues).

5.1 Convergence to Acceptable Allocations

In this section, we study load distribution properties for networks of heterogeneous contracts. We compare the results to those achieved using homogeneous contracts and show that our approach works well in heterogeneous systems. We also simulate fixed-price contracts and show

Contracts	Price Selection
Fixed	$\forall i \in S$: $\lvert\text{taskset}_i^F\rvert = 14$ operators $\forall i, j \in S$: $\text{price}(C_{i,j}) = \text{MC}(u, \text{taskset}_i^F - u)$
Range	$\forall i, j \in S$: $\text{max\_price}(C_{i,j}) = \text{MC}(u, \text{taskset}_i^F - u)$ $\text{min\_price}(C_{i,j}) = \text{max\_price}(C_{i,j}) - \Delta$ $\Delta = \delta_2(\text{taskset}_i^F)$ i.e., the range is [MC 12th op,MC 14th op]
Random	$\forall i \in S$: $\lvert\text{taskset}_i^F\rvert \in [12\text{ops}, 18\text{ops}]$ $\forall i, j \in S$ if $\lvert\text{taskset}_i^F\rvert \le \lvert\text{taskset}_j^F\rvert$ $\text{price}(C_{i,j}) = \text{MC}(u, \text{taskset}_i^F - u)$
Random Range	$\forall i, j \in S$: max price same as for Random $\text{min\_price}(C_{i,j}) = \text{max\_price}(C_{i,j}) - \Delta$ $\Delta = \delta_2(\text{taskset}_i^F)$

Desired result	Initial Load Allocation
underload	299 nodes with 22 ops 696 nodes with 7.7 ops on avg Overall average 12 ops/node
overload	299 nodes with 22 ops 696 nodes with 12.0 ops on avg Overall average 15 ops/node

Table 2: Simulated configurations. u is one operator. $\delta_2(\text{taskset}^F)$ is defined in Section 3.2.3.

that such contracts have good properties in practice, even though they do not always lead to acceptable allocations.

We thus study and compare the convergence properties of four variants of the mechanism: (1) Fixed, where all contracts set an identical fixed-price, (2) Range, where all contracts are uniform but define a small price-range, (3) Random Fixed, where contracts specify fixed but randomly chosen prices, and Random Range, where contracts define randomly selected price-ranges. Table 2 summarizes the price selection used for each type of contract. We limit price ranges to the variation in marginal cost resulting from moving only two tasks. This range is much smaller than required in theory to ensure acceptable allocations: the range is half the theoretical value for the smallest network diameter that we simulate (Table 1).

Starting from skewed load assignments, as summarized in Table 2, we examine how far the final allocation is from being acceptable. For an underloaded system, we measure the total excess load that nodes were unable to move (Figure 8). For an overloaded system, we measure the unused capacity that remains at different nodes (Figure 9)[5]. In both figures, the column with zero minimum number of contracts shows the excess load and available capacity that exist before any reallocation.

With the Fixed variant of the mechanism, we examine the properties of an underloaded system whose fixed contract prices are above the average load level and those of an overloaded system where prices are below average load. As shown in Figures 8 and 9, as the number of contracts increases, the quality of the final allocation improves: nodes manage to reallocate a larger amount of

[5]Each result is the average of nine simulations.

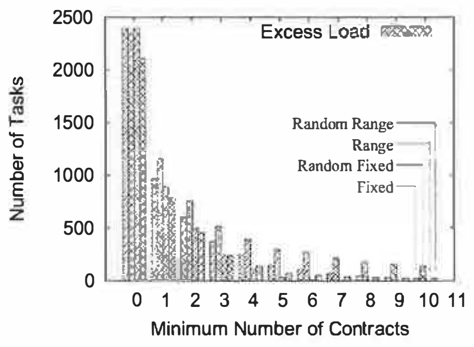


Figure 8: Excess load for the final allocation in an underloaded network of 995 nodes. First column shows excess load before any load movement.

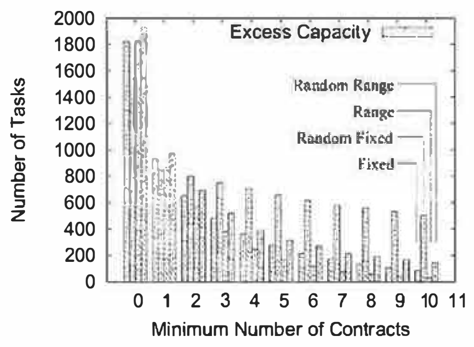
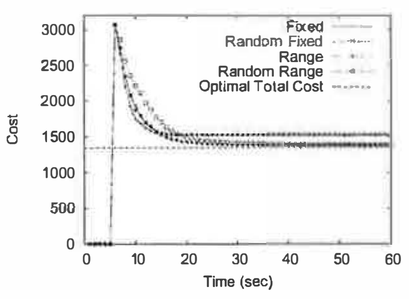


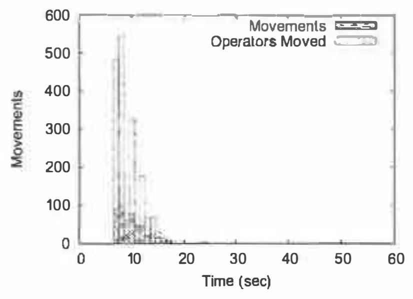
Figure 9: Left-over capacity for the final allocation in an overloaded network of 995 nodes.

excess load or exploit a larger amount of available capacity. With a minimum of 10 contracts, nodes successfully redistribute over 99% of excess load in the underloaded scenario and use over 95% of the initially available capacity in the overloaded case. The system thus converges to an allocation close to acceptable in both cases. Hence, even though they do not work well for specific configurations (as discussed in Section 3.2.3), fixed-price contracts can lead to allocations close to acceptable for randomly generated configurations.

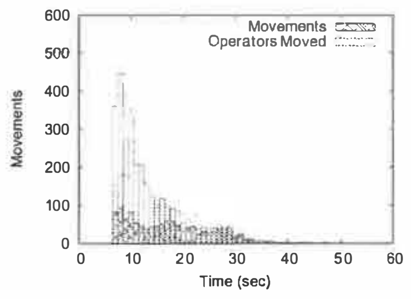
We re-run all simulations replacing fixed prices with a price range and observe the improvement in final allocation. We choose the price range to fall within the two load levels of 12 and 15 operators per node (Table 2, variant Range). The results, also presented in Figures 8 and 9, show that a minimum of seven contracts achieves an acceptable allocation. At that moment, the diameter of the network is five (Table 1) so the price-range is half the theoretically required value. When the system is overloaded, over 98% of available capacity is successfully used with a minimum of 10 contracts per node. Additionally, nodes below their capacity have room for only one additional operator. The final allocation is thus nearly acceptable, as defined in Section 3.2.3. This result shows that price



(a) Total cost.



(b) Number of movements with fixed-price contracts.



(c) Number of movements with price-range contracts.

Figure 10: Convergence speed in an underloaded network of 995 nodes.

ranges, even smaller than the theoretically computed values, suffice to achieve acceptable or nearly acceptable allocations in randomly generated configurations.

In practice, contracted prices will be heterogeneous. We thus simulate the same network of contracts again but with randomly selected prices (Random Fixed variant). To pick prices, each participant randomly selects a maximum capacity between 12 and 18 operators. When two participants establish a contract, they use the lower of their maximums as the fixed price. We find that random fixed-price contracts are less efficient than homogeneous contracts but they also achieve allocations close to acceptable (94% of excess load is re-allocated with 10 contracts as shown in Figure 8). Because we measure the capacity at each node as the number of operators that the node can accept before its marginal cost reaches its highest contracted price, when the number of contracts increases, so does the measured capacity. This in turn makes heterogeneous-price contracts appear less efficient than they actually are at using available capacity. The range of prices from which contracts are drawn is such that when each node has at least three contracts, the initially available capacity is roughly the same as with homogeneous contracts.

Finally, we explore heterogeneous price ranges by adding a lower price bound to each randomly chosen fixed price (Random Range variant). As shown in Figures 8 and 9, heterogeneous price-range contracts have similar properties to the uniform price-ranges. The final allocation is slightly worse than in the uniform case because nodes with small capacity impede load movements through chains and measured capacity increases with the number of contracts. We find, however, that in the underloaded case nodes were at most 2.1 operators above their threshold (average of multiple runs) and in the underloaded case nodes had at most capacity left for 2.5 operators. Heterogeneous prices thus lead to allocations close to acceptable ones.

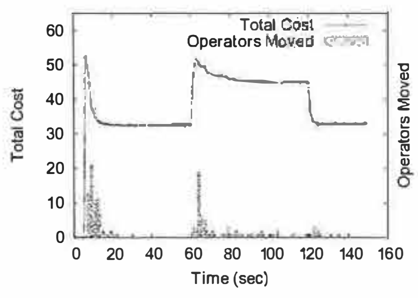
5.2 Convergence Speed

Figure 10(a) shows how the total cost decreases with time in the underloaded network of 995 nodes with a minimum of five contracts per node. In the simulation, each node tried to move load at most once every 2 seconds (no movements were allowed within the first 5 seconds of the simulation). For all variants, the cost decreases quickly and reaches values close to the final minimum within as few as 15 seconds of the first load movement. This fast convergence is partly explained by the ability of the bounded-price mechanism to balance load between any pair of nodes in a single iteration and partly by the ability of our mechanism to balance load simultaneously at many locations in the network. Fixed-price variants have a somewhat sharper decrease while also leading to a final allocation close to acceptable.

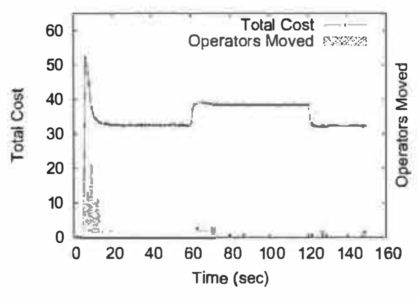
Figures 10(b) and (c) show the convergence speed measured as the number of load transfers (Movements) and the number of operators moved for homogeneous contracts (random prices produce almost identical trends). For both variants, the first load movements re-allocate many operators, leading to the fast decrease in the total cost in early phases of the convergence. Fixed-price contracts converge faster than price-range contracts because more operators can be moved at once. The convergence also stops more quickly but, as shown above, the allocation is slightly worse than with a small price range.

5.3 Stability under Changing Load

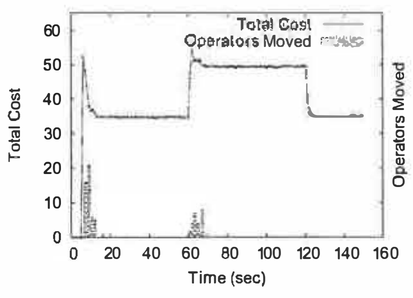
Next, we examine how the mechanism responds to sudden step-shifts in load. A bad property would be for small step shifts in load to lead to excessive re-allocations. We subject a network of 50 nodes (with a minimum of three fixed or three price-range contracts per node) to a sudden load increase (at time 60 sec) and a sudden load decrease (at time 120 sec). We run two series of experiments. We first create a large variation with 30% extra



(a) Large load variation with price range: 30% extra load.



(b) Small load variation with price range: 15% extra load.



(c) Large load variation with fixed prices: 30% extra load.

Figure 11: Assignment stability under variable load. Load added at 60 sec and removed at 120 sec.

load (10 extra operators each to 10 different nodes). We then repeat the experiment adding only 15% extra load. Figure 11 shows the total cost and operator movements registered during the simulation.

A 30% load increase, concentrated around a few nodes, makes these nodes exceed their capacity leading to a few re-allocations both with fixed prices and price ranges. Fixed prices, however, produce fewer re-allocations because convergence stops faster. When load is removed, spare capacity appears. If nodes use a price range, a small number of re-allocations follows. With fixed-prices, since nodes all run within capacity before the load is removed, nothing happens. The 15% load increase leads to an almost insignificant number of movements even when a small price-range is used. Indeed, fewer nodes exceed their capacity and load variations within capacity do not lead to re-allocations. Both variants of the mechanism thus handle load variations without excessive re-allocations.

5.4 Prototype Experiments

We evaluate our prototype on the network intrusion detection query (with 60 sec windows and without the final join) running on network connection traces collected at MIT (1 hour trace from June 12, 2003) and at an ISP in Utah (1 day trace from April 4, 2003). To reduce the possible granularity of load movements, we partition the Utah log into four traces that are streamed in parallel, and the MIT log into three traces that are streamed in parallel. To increase the magnitude of the load, we play the Utah trace with a 20× speed-up and the MIT trace with an 8× speed-up.

Figure 12 illustrates our experimental setup. Node 0 initially processes all partitions of the Utah and MIT traces. Nodes 1 and 2 process 2/3 and 1/3 of the MIT trace, respectively. Node 0 runs on a desktop with a Pentium(R) 4, 1.5GHz and 1GB of memory. Nodes 1 and 2 run on a Pentium III TabletPC with 1.33GHz and 1GB of memory. The nodes communicate over a 100 Mbps Ethernet. All clients are initially on the same machines as the nodes running their queries. All Medusa nodes have fixed-price contracts with each other and are configured to take or offer load every 10 seconds.

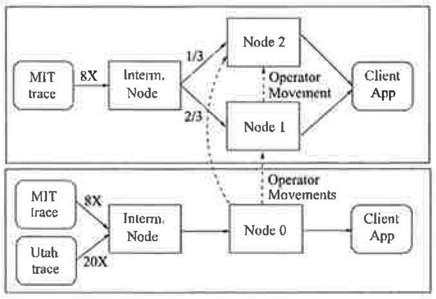


Figure 12: Experimental setup.

Figure 13 shows the results obtained. Initially, the load at each node is approximately constant. At around 650 seconds (1) the load on the Utah trace starts increasing and causes Node 0 to shed load to Node 1, twice (on Figure 13 these movements are labeled (2) and (3)). After the second movement, load increases slightly but Node 1 refuses additional load making Node 0 move some operators to Node 2 (4). The resulting load allocation is not uniform but it is acceptable. At around 800 seconds (5), Node 1 experiences a load spike, caused by an increase in load on the MIT trace. The spike is long enough to cause a load movement from Node 1 to Node 2 (6), making all nodes operate within capacity again. Interestingly, after the movement the load on Node 1 decreases. This decrease does not cause further re-allocations as the allocation remains acceptable.

In our experimental setup, it takes approximately 75 ms to move a query fragment between two nodes. Each movement proceeds as follows. The origin node sends to the remote node a list of operators and stream subscrip-

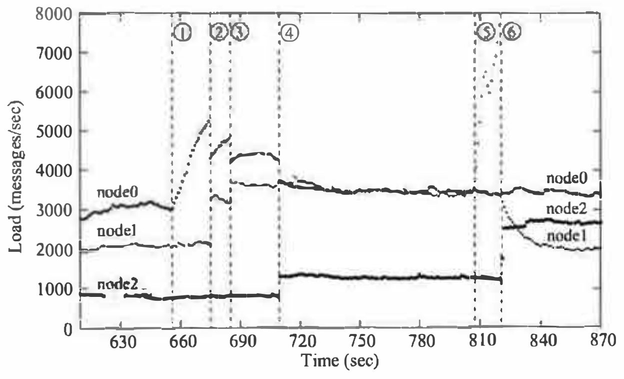


Figure 13: Load at three Medusa nodes running the network intrusion detection query over network connection traces.

tions (i.e., a list of client applications or other Medusa nodes currently receiving the query output streams). The remote node instantiates the operators locally, subscribes itself to the query input streams, starts the query, and sets up the subscriptions to the output streams. After the remote query starts, the origin node drains accumulated tuples and deletes the query. Both nodes update the catalog asynchronously. When a query moves, client applications see a small number of duplicate tuples because the new query starts before the old one stops. They may also see some reordering if the origin node was running behind before the move.

In our current implementation, we do not send the state of operators to the remote location. This approach works well for all stateless operators such as *filter*, *map*, and *union* as well as for operators that process windows of tuples without keeping state between windows (e.g., *windowed joins* and some types of *aggregates*). For these latter operators, a movement disrupts the computation over only one window. For more stateful operators, we should extend the movement protocol to include freezing the state of the original query, transferring the query with that state, and re-starting the query from the state at the new location. We plan to explore the movements of stateful operators in future work.

6 Conclusion

In this paper, we presented a mechanism for load management in loosely coupled, federated distributed systems. The mechanism, called the *bounded-price mechanism*, is based on pairwise contracts negotiated offline between participants. These contracts specify a bounded range of unit prices for load transfers between partners. At runtime, participants use these contracts to transfer excess load at a price within the pre-defined range.

Small price-ranges are sufficient for the mechanism to produce acceptable allocations in an underloaded network of uniform nodes and contracts, and produce allocations close to acceptable in other cases. Compared to previous approaches, this mechanism gives participants control and privacy in their interactions with others. Contracts allow participants not only to constrain prices but also practice price discrimination and service customization. The approach also has a low runtime overhead.

Participants have flexibility in choosing contract prices. We show that even randomly chosen prices from a wide range achieve allocations close to acceptable. We suggest, however, that participants first negotiate relatively high fixed-price contracts to maximize their chances of shedding excess load while minimizing runtime overhead and only later negotiate additional contracts with lower prices. Additionally, if participants notice that they often stand between overloaded and underloaded partners, they should re-negotiate some of their contracts to cover a small price-range and make a small profit by forwarding load from their overloaded to their underloaded partners.

Although the load management mechanism introduced in this paper is motivated by federated distributed stream processing, it also applies to other federated systems such as Web services, computational grids, overlay-based computing platforms, and peer-to-peer systems.

In this paper, we did not address high availability. Because each participant owns multiple machines, participant failures are rare. We envision, however, that if the participant running the tasks fails, it is up to the original nodes to recover the failed tasks. If the original participant fails, though, the partner continues processing the tasks until the original participant recovers. Contracts could also specify availability clauses. We plan to investigate high availability further in future work.

Acknowledgments

We thank Rahul Sami for many invaluable suggestions. We thank Jaeyeon Jung and Dave Andersen from MIT, and Noah Case from Syptec, for providing us the network connection traces.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The Int. Journal on Very Large Data Bases*, Sept. 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of 2002 ACM Symposium on Principles of Database Systems*, June 2002.

[3] P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. Technical Report HPL-98-203, Hewlett-Packard Company, 1998.

[4] R. Buuya, H. Stockinger, J. Giddy, and D. Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proc. of SPIE Int. Symposium on The Convergence of Information Technologies and Communications (ITCom 2001)*, Aug. 2001.

[5] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD Int. Conference on Management of Data*, May 2000.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[8] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[9] B. N. Chun. *Market-Based Cluster Resource Management*. PhD thesis, University of California at Berkeley, 2001.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[11] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. of the ACM SIGCOMM 2003 Conference*, Aug. 2003.

[12] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proc. of the 21st Symposium on Principles of Distributed Computing*, July 2002.

[13] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63:21–41, 2001.

[14] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. of the 6th Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Sept. 2002.

[15] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientist, Jan. 1996.

[16] I. T. Foster and C. Kesselman. Computational grids. In *Proc. of the Vector and Parallel Processing (VECPAR)*, June 2001.

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[18] Y. Fu and A. Vahdat. Service level agreement based distributed resource allocation for streaming hosting systems. In *Proc. of 7th Int. Workshop on Web Content Caching and Distribution*, Aug. 2002.

[19] R. Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communication*, COM-25(1), Jan. 1977.

[20] M. Jackson. Mechanism theory. *Forthcoming in Encyclopedia of Life Support Stystems*, 2000.

[21] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for Web services. Technical Report RC22456, IBM Corporation, May 2002.

[22] J. F. Kurose. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[23] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[24] W. Lehr and L. W. McKnight. Show me the money: Contracts and agents in service level agreement markets. `http://itc.mit.edu/itel/docs/2002/show_me_the_money.pdf`, 2002.

[25] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. *The Ecology of Computation*, 1988.

[26] Mesquite Software, Inc. CSIM 18 user guide. `http://www.mesquite.com`.

[27] M. S. Miller and K. E. Drexler. Markets and computation: Agoric open systems. In B. Huberman, editor, *The Ecology of Computation*. Science & Technology, 1988.

[28] C. Ng, D. C. Parkes, and M. Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[29] C. Ng, D. C. Parkes, and M. Seltzer. Virtual worlds: Fast and strategyproof auctions for dynamic resource allocation. `http://www.eecs.harvard.edu/~parkes/pubs/virtual.pdf`, June 2003.

[30] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[31] N. Nisan and A. Ronen. Computationally feasible VCG mechanisms. In *Proc. of the Second ACM Conference on Electronic Commerce (EC00)*, Oct. 2000.

[32] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35, 2001.

[33] D. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency (Chapter 2)*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[34] D. Parkes. Price-based information certificates for minimal-revelation combinatorial auctions. *Agent Mediated Electronic Commerce IV*, LNAI 2531:103–122, 2002.

[35] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the First Workshop on Hot Topics in Networks*, Oct. 2002.

[36] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 13th Conference on Systems Administration (LISA-99)*, Nov. 1999.

[37] A. Sahai, A. Durante, and V. Machiraju. Towards automated SLA management for Web services. Technical Report HPL-2001-310R1, Hewlett-Packard Company, July 2001.

[38] A. Sahai, S. Graupner, V. Machiraju, and A. van Moorsel. Specifying and monitoring guarantees in commercial grids through SLA. Technical Report HPL-2003-324, Hewlett-Packard Company, Nov. 2002.

[39] T. W. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proc. of the 12th Int. Workshop on Distributed Artificial Intelligence*, pages 295–308, 1993.

[40] T. W. Sandholm. Contract types for satisficing task allocation: I theoretical results. In *AAAI Spring Symposium Series: Satisficing Models*, Mar. 1998.

[41] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th Int. Conference on Data Engineering (ICDE 2003)*, Mar. 2003.

[42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM 2001 Conference*, pages 149–160, Aug. 2001.

[43] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal: The Int. Journal on Very Large Data Bases*, 5, Jan. 1996.

[44] The Condor Project. Condor high throughput computing. `http://www.cs.wisc.edu/condor/`.

[45] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[46] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, SE-18(2):103–117, 1992.

Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval

Chunqiang Tang and Sandhya Dwarkadas
Computer Science Department, University of Rochester
{sarrmor,sandhya}@cs.rochester.edu

Abstract

Content-based full-text search still remains a particularly challenging problem in peer-to-peer (P2P) systems. Traditionally, there have been two index partitioning structures—partitioning based on the document space or partitioning based on keywords. The former requires search of every node in the system to answer a query whereas the latter transmits a large amount of data when processing multi-term queries. In this paper, we propose *eSearch*—a P2P keyword search system based on a novel hybrid indexing structure. In eSearch, each node is responsible for certain terms. Given a document, eSearch uses a modern information retrieval algorithm to select a small number of top (important) terms in the document and publishes the *complete* term list for the document to nodes responsible for those top terms. This *selective* replication of term lists allows a multi-term query to proceed local to the nodes responsible for query terms. We also propose automatic query expansion to alleviate the degradation of quality of search results due to the selective replication, overlay *source* multicast to reduce the cost of disseminating term lists, and techniques to balance term list distribution across nodes.

eSearch is scalable and efficient, and obtains search results as good as state-of-the-art centralized systems. Despite the use of replication, eSearch actually consumes less bandwidth than systems based on keyword partitioning when publishing metadata for a document. During a retrieval operation, it searches only a small number of nodes and typically transmits a small amount of data (3.3KB) that is independent of the size of the corpus and grows slowly (logarithmically) with the number of nodes in the system. eSearch's efficiency comes at a modest storage cost, 6.8 times that of systems based on keyword partitioning. This cost can be further reduced by adopting index compression or pruning techniques.

1 Introduction

Peer-to-Peer (P2P) systems have gained tremendous interest from both the user and research community in the past several years. First-generation systems such as Gnutella and KaZaA are already prevalent, and second-generation systems such as PAST [32] and CFS [14] based on Distributed Hash Tables (DHTs) are under serious development (e.g., the IRIS project [19]). With a gigantic amount of information in these systems, it would be impossible for users to remember or even know the place or precise ID of the desired data. The capability to retrieve documents using content-based full-text search would greatly improve the usability of these systems.

Although it is possible to build a dedicated search engine to index contents in P2P systems in a way similar to how Google indexes the Web, a P2P search system built on top of the same nodes already used in the P2P storage system is particularly attractive because of its low cost, ease of deployment, availability, and scalability. In this paper, we study the challenging problem of building P2P keyword search systems.

To facilitate the retrieval of documents, a distributed (not necessarily P2P) search system places information regarding the occurrence of terms (words or phrases) in documents in the form of *metadata* at certain places in the system. The metadata placement strategy in existing systems are based on either *local* or *global* indexing [42].

In local indexing (see Figure 1 (i)), metadata are partitioned based on the document space. The complete term list of a document is stored on a node. A term list $X \dashrightarrow a, c$ means that document X contains terms a and c. During a retrieval operation, the query is broadcast to all nodes. Since a node has the complete term list for documents that it is responsible for, it can compute the relevance between the query and its documents without consulting others. The drawback, however, is that every node is involved in processing every query, rendering systems of this type unscalable. Gnutella and search engines such as AllTheWeb (www.alltheweb.com) [30] are based on variants of local indexing.

In global indexing (see Figure 1 (ii)), metadata are distributed based on terms. Each node stores the complete inverted list of some terms. An inverted list $a \dashrightarrow X, Z$ indicates that term a appears in document X and Z. To answer a query consisting of multiple terms, the query is sent to nodes responsible for those terms. Their inverted lists are transmitted over the network so that a join to identify documents that contain multiple query terms can be performed. The communication cost for a join grows proportionally with the length of the inverted lists, i.e., the size of the corpus. Most recent proposals for P2P keyword search [16, 20, 28, 39] are based on global indexing, but with enhancements to reduce the communication cost, for instance, using Bloom filters to summarize the inverted lists or incrementally transmitting the inverted lists and terminating early if sufficient results have already been obtained. In the following, we will simply refer to these systems as *Global-P2P* systems.

Challenging conventional wisdom that uses either local or global indexing, we propose a *hybrid indexing* structure to combine their benefits while avoiding their limitations. The basic tenet of our approach is *selective* metadata replication. Like global indexing, hybrid indexing distributes metadata based on terms (see Figure 1 (iii)). Each node j is responsible for the inverted list of some term t. In addition, for each document D in the inverted list for term t, node j also stores the complete term list for document D. Given a multi-term query, the query is sent to nodes responsible for those terms. Each of these nodes then does a local search without consulting others, since it has the complete term list for documents in its inverted list. Our system based on hybrid indexing is called *eSearch*. It uses a Distributed Hash Table (DHT) to map a term to a node where the inverted list for the term is stored. We chose Chord [38] for eSearch, but other DHTs such as Pastry, CAN, and Tapestry can also be used without major changes to our design.

When naively implemented, eSearch's search efficiency obviously comes at the expense of publishing more metadata, requiring more communication and storage. We propose several optimizations that reduce communication by up to 97% and storage by up to 90%, compared with a naive hybrid indexing. Below we outline one important optimization—top term selection.

Each document contains many words. Some of them are central to ideas described in the document while the majority are just auxiliary words. Modern statistical information retrieval (IR) algorithms such as vector space model (VSM) [33, 36] assign a weight to each term in a document. Terms central to a document are automatically identified by a heavy weight. In eSearch, we only publish the term list for a document to nodes responsible for top (important) terms in that document. Figure 1 (iv) illustrates this optimization. Document X contains terms a and c but its term list is only published to computer 3 since only term c is important in X.

This optimization, however, may degrade the quality of search results. A query on a term that is not among the top terms of a document cannot find this document.

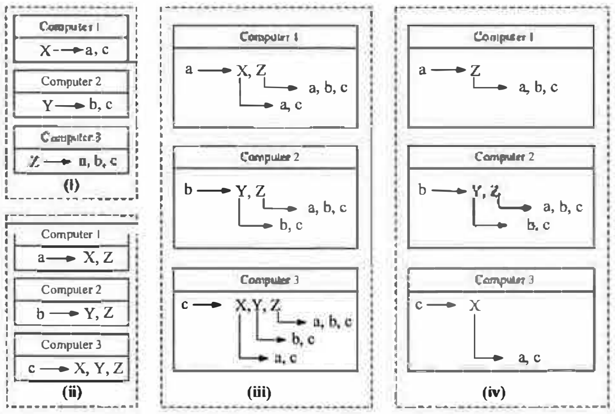


Figure 1: Comparison of distributed indexing structures. **(i)** Gnutella-like local indexing. **(ii)** Global indexing. **(iii)** Hybrid indexing. **(iv)** Optimized hybrid indexing. *a*, *b*, and *c* are terms. *X*, *Y*, and *Z* are documents. This example distributes metadata for three documents (*X*-*Z*) that contain terms from a small vocabulary (*a*-*c*) to three computers (1-3). Term list $X \dashrightarrow a, c$ means that document X contains term a and c. Inverted list $a \dashrightarrow X, Z$ indicates that term a appears in document X and Z.

Our argument is that, if none of the query terms is among the top terms for a document, IR algorithms are unlikely to rank this document among the best matching documents for this query anyway. Thus, the top search results for this query are unlikely to be affected by skipping this document. In Section 3, we quantify the precision degradation due to this optimization. Our results show that eSearch obtains search quality as good as the centralized baseline by publishing a document under its top 20 terms.

In order to further reduce the chance of missing relevant documents, we adopt automatic query expansion [23]. We draw on the observation that, with more terms in a query, it is more likely that a document relevant to this query is published under at least one of the query terms. This scheme automatically identifies additional terms relevant to a query and also searches nodes responsible for those terms. We also propose an overlay *source* multicast protocol to efficiently disseminate term lists, and two decentralized techniques to balance the distribution of term lists across nodes.

We evaluate eSearch through simulations and analysis. The results show that, owing to the optimization techniques, eSearch is scalable and efficient, and obtains search results as good as the centralized baseline. Despite the use of metadata replication, eSearch actually consumes less bandwidth than the Global-P2P systems when publishing a document. During a retrieval operation, eSearch typically transmits 3.3KB of data. These costs are independent of the size of the corpus and grow slowly (logarithmically) with the number of nodes in the system. eSearch's efficiency comes at a modest storage cost (6.8 times that of the Global-P2P systems), which can be further reduced by adopting index compression [43] or pruning [7].

Given the quickly increasing capacity and decreasing price of disks, we believe trading modest disk space for communication and precision is a proper design choice for P2P systems. According to Blake and Rodrigues [2], in 15 years, disk capacity increased by 8000-fold while bandwidth for an end user increased by only 50-fold. Moreover, work in the Farsite project [3] observed that about 50% of disk space on desktops was not in use.

In this paper, our focus is on designing an indexing architecture to support efficient P2P search. Under this architecture, we currently use Okapi [31, 36] (a state-of-the-art content-based IR algorithm) to rank documents. In reality, search engines combine many IR techniques to rank documents. The adoption and evaluation of those techniques in our architecture is a subject of future work.

The remainder of the paper is organized as follows. Section 2 provides an overview of eSearch's system architecture. Sections 3 to 5 describe and evaluate individual pieces of our techniques, including top term selection and automatic query expansion (Section 3), overlay source multicast (Section 4), and balancing term list distribution (Section 5). We analyze eSearch's system resource usage and compare it with Global-P2P systems in Section 6. Related work is discussed in Section 7. Section 8 concludes the paper.

2 System Architecture

Figure 2 depicts eSearch's system architecture. A large number of computers are organized into a structured overlay network (Chord [38]) to offer IR service. Nodes in the overlay collectively form an eSearch *Engine*. Inside the Engine, nodes have completely homogeneous functions. A client (e.g., node X) intending to use eSearch connects to any Engine node (e.g., node E) to publish documents or submit queries. Engine nodes are also user nodes that can initiate document publishing or a search on behalf of their user.

Among nodes in the P2P system, we intentionally distinguish server-like Engine nodes that are stable and have good Internet connectivity from the rest. Excluding ephemeral nodes from the Engine avoids unnecessary maintenance operations. Moreover, not even every stable node needs to be included in the Engine, so long as the Engine has enough capacity to offer the desired level of quality of service.

When a new node joins the P2P system, it starts as a client. After being stable for a threshold time (e.g., 20 minutes) and if the load inside the Engine is high, it joins the Engine to take over some load, using the protocol

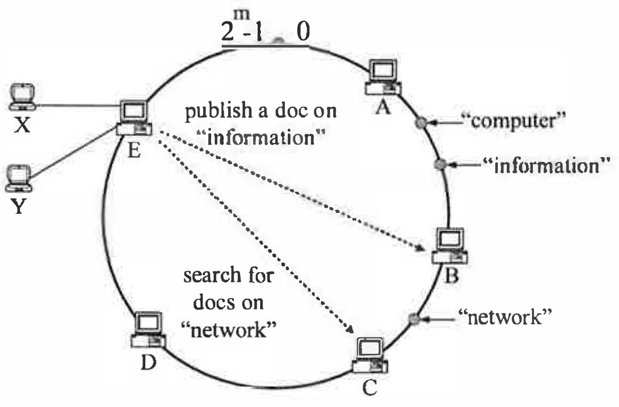


Figure 2: System architecture of eSearch.

described in Section 5. Data stored on an Engine node are replicated on its neighbors. Should a node fail, one of its neighbors will take over its job seamlessly.

Inside the Engine, nodes are organized into a ring topology corresponding to an ID space ranging from 0 to $2^m - 1$ where $m = 160$. Each node is assigned an ID drawn from this ID space, and is responsible for the key range between its ID and the ID of the previous node on the ring. Each term is hashed into a key in the ID space. The node whose ID immediately follows the term's key is responsible for the term. For instance, node B is responsible for inverted lists for the term "computer" and "information". Lookup for the node responsible for a given key is done through routing in the overlay. With the help of additional links not shown in Figure 1, Chord on average routes a message to its destination in $O(\log N)$ hops, where N is the number of nodes in the overlay.

Document metadata are organized based on the hybrid indexing structure. To publish a document, a client sends the document to the Engine node that it connects to. The Engine node identifies top (important) terms in the document and disseminates its term list to nodes responsible for those top terms, using overlay source multicast to economize on network bandwidth (see Section 4).

A client X starts a search by submitting a query to the Engine node E that it connects to, which then uses overlay routing to forward the query to nodes responsible for terms in the query. Those nodes do a local search, identifying a small number of best matching documents, and return the ID and relevance score (a numerical value that specifies relevance between a document and a query) of those documents to node E. Node E gives returned documents a global rank based on the relevance score and presents the top documents to client X. To improve search quality, node E may expand the query with more relevant terms learned from returned documents, start a second round of search, and then present the final results to client X (see Section 3.2).

To avoid processing the same query repeatedly and also to alleviate hot spots corresponding to popular queries, query results are cached for a certain amount of time at the nodes processing the query and the paths along which the query is forwarded. If a query arrives at a node with live cached results, those results are returned immediately.

3 Top Term Selection and Automatic Query Expansion

In Section 1, we proposed hybrid indexing for efficient P2P search, by combining the advantages of local and global indexing while avoiding their limitations. Like global indexing, it distributes metadata based on terms. Like local indexing, it stores the complete term list for a document on a node. Given a query, it searches only a small number of nodes and avoids transmitting inverted lists over the network, thereby supporting efficient search. The drawback, however, is that it publishes more metadata, requiring more storage and potentially more communication.

Our first optimization is to avoid publishing a document under *stopwords*—words that are too common to have real effect on differentiating one document from another, e.g., the word "the". This simple optimization results in great savings since as much as 50% of a document's content is stopwords. Our second optimization is to use vector space model (VSM) [36] to identify top (important) terms in a document, and only publish its term list to nodes responsible for those terms.

3.1 Top Term Selection

We first provide an overview of VSM. VSM assigns a weight to each term in a document or query. Terms central to a document are automatically identified by a heavy weight. VSM computes the relevance between a document D and a query Q as

$$\text{relevance}(D, Q) = \sum_{t \in D, Q} d_t \cdot q_t \qquad (1)$$

where t is a term appearing in both document D and query Q, d_t is term t's weight in document D, and q_t is term t's weight in query Q. Documents with the highest relevance score are returned as search results. The weight of a term is decided by several factors, including the length of the document, the frequency of the term in the document, and the frequency of the term in other documents. Intuitively, if a term appears in a document with a high frequency, there is a good chance that the term could be used to differentiate the document from others. However, if the term also appears in many other documents, its importance should be penalized.

A myriad of term weighting schemes have been proposed, among which Okapi [31, 36] has been shown to be particularly effective. For instance, among the eight systems that achieved the best performance in the TREC-8 *ad hoc* track [41], five of them were based on Okapi. We adopt Okapi in eSearch but omit its details here due to space limitations. Okapi relies on some global statistics (e.g., the popularity of terms) to compute term weights. Previous work [17] has shown that statistical IR algorithms can work well with estimated statistics. eSearch uses a combining tree to sample documents, merge statistics, and disseminate the combined statistics. In the following, we assume the global statistics are known. An evaluation of eSearch's sensitivity to the estimated statistics is the subject of ongoing work.

We conduct experiments on volumes 4 & 5 of the TREC corpus [41] to determine if it is true that some terms in a document are much more important than others. The TREC corpus is a standard benchmark widely used in the IR community. It comes with a set of carefully constructed queries and manually selected relevant documents for each query, against which one can quantitatively evaluate the search quality of a system. It includes 528,543 documents from the news, magazines, congressional records, etc. The average length of a document is 3,778 bytes.

Cornell's SMART system [5] implements a framework for VSM. We extend it with an implementation of Okapi and use it to index the TREC corpus. SMART comes with a list of 571 stopwords, which is used as is in our experiments. The SMART stemmer is used without modification to strip word endings (i.e., "book" and "books" are treated as the same).

For each document, we sort its terms by decreasing Okapi weight and compute the relative weight of each term to the biggest term weight in the document. We average this normalized term weight across all documents, computing a mean for each term rank, and report these means in Figure 3. Note that the Y axis is in log scale. The normalized term weight largely follows a Zipf distribution except that the weight of the top 20 terms drops even faster than a Zipf distribution. This confirms our intuition that a small number of terms are much more important than others in a document. One analogy is that these terms are words in the "Keyword" section of a paper, except that eSearch extracts them automatically.

3.2 Automatic Query Expansion

Only publishing a document under its top terms may degrade the quality of search results. A query on a term that is not among the top terms of a document cannot find this document. We adopt automatic query expansion [23] (also called automatic relevance feedback) to

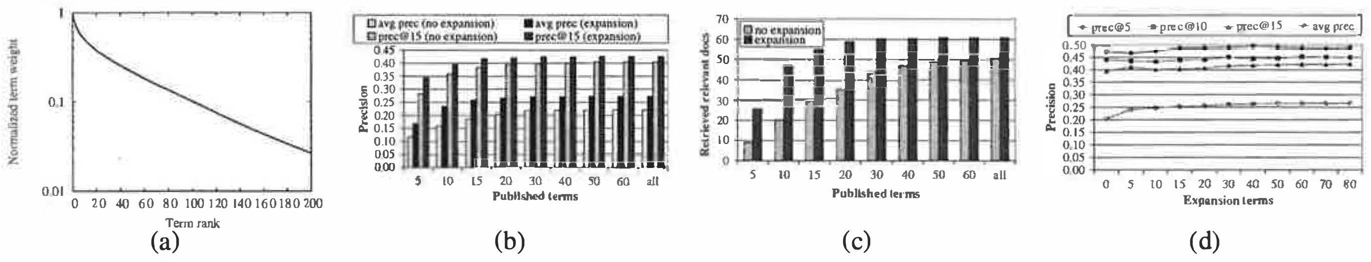


Figure 3: (a) Ranked term weight of the TREC corpus, normalized to the biggest term weight in each document. (b) eSearch's precision with respect to the number of terms under which a document is published. The performance of the "all" series is equivalent to that of a centralized system. (c) The average number of retrieved relevant documents for a query when returning 1,000 documents. (d) eSearch's precision with respect to the number of expanded query terms.

alleviate this problem. We draw on the observation that, with more terms in a query, it is more likely that a document relevant to this query is published under at least one of the query terms. This scheme automatically expands a short query with additional relevant terms. It has been show as one important technique to improve performance in centralized IR systems [36].

We experimented with several query expansion techniques and found that complex ones such as [23] only marginally improve search quality for the TREC corpus compared with simple ones. For the sake of clarity we describe below a simple scheme that is degenerated from [23] but has the same performance.

Given a query, eSearch first uses the hybrid indexing structure to retrieve a small number f of best matching documents. We call these documents *feedback documents*. For each term in the feedback documents, the Engine node that starts the search on behalf of a client computes the average weight of terms in the feedback documents and chooses k terms that have the biggest average weight. These terms are assumed to be relevant to the query and are added into the query. The new query is then used to retrieve the final set of documents for return. Recall that VSM assigns a weight for query terms as it does for document terms (see Equation 1). The weight for an expanded query term, which is not assigned by VSM, is its average weight in feedback documents divided by a constant α. Through experiments we found that $f = 10$ and $\alpha = 16$ turned out to work well. Alternatively, one may set α to a very large number to make the overall ranking equivalent to that without the expanded query terms, but still search nodes corresponding to expanded terms to reduce the chance of missing relevant documents.

We illustrate these steps through an example. Suppose VSM identifies "routing" as the only important term for a document D. Given a query of "computer network", eSearch first retrieves relevant documents with either "computer" or "network" as one of their important terms. After a look at the retrieved documents, eSearch finds that "routing" seems to be an important common word among them. It then expands the query as "computer network routing", assigning "routing" a relatively smaller weight than the weight for the original query terms "computer network". The new query is then used to retrieve a final set of documents. This time, it can find document D.

3.3 Experimental Results

We experiment with the TREC corpus to determine proper parameters for eSearch, including the number of top terms under which a document is published and the number of expanded terms for a query. We use the "title" field of TREC topics 351-450 as queries. On average, each query consists of 2.4 terms and has 94 manually identified relevant documents in the corpus. Note that this identification is subjective and based on user input, which implies that a document that contains all terms in a query is not necessarily relevant to the query, and a document relevant to a query need not contain all terms in the query.

The metric to quantify the quality of search results is *precision*, defined as the number of retrieved relevant documents divided by the number r of retrieved documents. For instance, $prec@15$=0.4 means that, when returning 15 documents for a query, about 40% of the returned documents will be evaluated by users as really relevant to the query. $prec@r$ varies with r. The *average precision* for a single topic is the mean of the precision obtained after each relevant document is retrieved (using zero as the precision for relevant documents that are not retrieved). We are particularly interested in high-end precision (e.g., $prec@15$) as users usually only view the top 10 search results.

Figure 3 (b) reports eSearch's precision with respect to the number of terms under which a document is published (shown on the X axis). The "expansion" and "no expansion" series are with and without query expansion, respectively. In this experiment, we set the number of expanded terms to 30. "All" means that a document is published under all its terms, whose precision is equal to that of a centralized IR system. Two observations can be drawn from this figure. (1) eSearch can approach the precision of the centralized baseline by publishing a document under a small number of selected terms, e.g., top 20 terms. (2) Automatic query expansion improves precision, particularly when documents are published under very few top terms.

We next examine the performance when returning a large number of documents for each query. This is the case when a user wishes to retrieve as many relevant documents as possible. Figure 3 (c) reports the average number of retrieved relevant documents for a query when returning 1,000 documents. Query expansion increases the number of retrieved relevant documents by 22%. The performance of the "expansion" series catches up with that of the centralized baseline earlier than the "no expansion" series, showing that the use of query expansion allows eSearch to publish documents under fewer terms to achieve the performance of centralized systems.

Figure 3 (d) shows the precision with respect to the number of expanded query terms. In this experiment, each document is published under its top 20 terms. The high-end precision is not very sensitive to query expansion. The average precision improves slowly after the number of expanded terms exceeds 10. Adding more terms into a query results in searching more nodes. This figure indicates that the benefit of query expansion can be reaped with limited overhead in eSearch.

Comparing the "no expansion" series in Figures 3 (b) and (c), we find that top search results are relatively insensitive to top term selection whereas the low-rank search results are affected more severely. Accordingly, query expansion is most useful for improving low-end precision but not for high-end precision. This observation leads to a further optimization. Given a new query, eSearch first retrieves a small number of documents without query expansion. If the user is unsatisfied with the results, eSearch uses these documents as feedback documents to expand the query and do a second round of search to return more documents.

Figures 3 (b) to (d) show that eSearch's average retrieval quality is as good as the centralized baseline. In Table 1 we try to further understand the difference for individual queries between eSearch and the baseline. When documents are published under a few terms, eSearch performs badly for a few queries. For instance, when publishing documents under only their top 5 terms, there is one query that eSearch finds no relevant documents for whereas the baseline can find 9 relevant documents. When the number of selected top terms increases, eSearch's performance improves quickly. When publishing documents under their top 20 terms, the worst relative performance for eSearch is one query for which eSearch retrieves four fewer relevant documents.

difference	top 5 terms	top 10 terms	top 20 terms
d=3	1	0	0
d=2	3	2	0
d=1	5	3	3
d=0	53	73	90
d=-1	15	9	5
d=-2	6	8	0
d=-3	4	3	1
d=-4	3	0	1
d=-5	3	1	0
d=-6	3	0	0
d=-7	3	1	0
d=-8	0	0	0
d=-9	1	0	0

Table 1: Difference in the number of retrieved relevant documents between eSearch and the centralized baseline when returning 10 documents for each query. The columns correspond to eSearch with different configurations. One entry with value p in the "d=k" row (e.g., the entry with value 8 in the "d=-2" row) means that, out of the 100 TREC queries, p queries return k more relevant documents in eSearch than in the baseline (or return fewer relevant documents if $k < 0$). For instance, when publishing documents under their top 10 terms, for 8 queries, eSearch returns 2 fewer relevant documents than the baseline, and for 73 queries, eSearch returns the same number of relevant documents as the baseline. For some queries, eSearch does better than the baseline because of the inherent fuzziness in Okapi's ranking function (i.e., just focusing on important terms may actually improve retrieval quality sometimes).

3.4 Discussions

Our evaluation so far has assumed that eSearch always publishes documents under the same number of top terms. This number can actually be varied from document to document. Intuitively, it may be more reasonable to publish long documents under more terms. Our current implementation includes a heuristic that publishes documents with big term weights under more terms. A detailed discussion is omitted due to space limitations.

The worst case scenario for eSearch is that a query is about a term that is not important in *any* document, e.g., the phrase "because of". eSearch cannot return any result although documents containing this term do exist. Although this scenario does exist in theory, in practice, eSearch's search quality on the widely used TREC benchmark corpus is as good as the centralized baseline. To achieve good retrieval quality, the absolute number of selected top terms may vary from corpus to corpus, but we expect it to be a small percentage of the total number of unique terms in a document, as the importance of terms in a document decreases quickly (see Figure 3 (a)). The same trend also holds for several other copora we tested, including MED, ADI, and CRAN (available in the SMART package [5]). Moreover, it is always possible to flood a hard query to every node, degrading eSearch's efficiency to Gnutella in really rare cases.

It should be emphasized that eSearch is not tied to any particular document ranking algorithm. For documents with cross reference links, link analysis algorithms such as Google's PageRank [4] can be incorporated, for instance, by combining PageRank with VSM to assign term weights [21] or publishing important Web pages (identified by PageRank) under more terms.

4 Disseminating Document Metadata

Given a new document, eSearch uses Okapi to identify its top terms and distributes its term list to nodes responsible for those terms. Since the same data are sent to multiple recipients, one natural way to economize on bandwidth is to multicast the data to the recipients.

Due to a variety of deployment issues, IP multicast is not widely supported in the Internet. Instead, several overlay multicast systems have been recently proposed, e.g., Narada [9]. These systems usually target multimedia or content distribution applications, where a multicast session is long. Thus, they can amortize the overhead for network measurement, multicast tree creation, adaptation, and destruction, over a long data session.

The scenario for term list dissemination, however, is quite different. Typically, a document is only several kilobytes and its term list is even smaller. As a result, eSearch disseminates data through a large number of extremely short sessions. Although the absolute saving from multicast in a single session is small, the aggregate savings over a large number of sessions would be huge. But this requires protocol overhead for multicast tree creation and destruction to be very small.

To this end, we propose overlay *source* multicast to distribute term lists. It does not use costly messaging to explicitly construct or destroy the multicast tree. Assisted by Internet distance estimation techniques such as GNP [25], the data source locally computes the structure of an efficient multicast tree that has itself as the root and includes all recipients. It builds an application-level packet with the data to be disseminated as payload and the structural information of the multicast tree as header, which specifies the IP addresses of the recipients and the parent-child relationship among them. It then sends the packet to the first-level children in the multicast tree. A recipient of this packet inspects the header to find its children in the multicast tree, strips off information for itself from the header, and forwards the rest of the packet to its children, and so forth.

In the following, we provide more details on the method that the data source uses to compute the structure of the multicast tree. Our method is based on GNP. A node using GNP measures RTTs to a set of well-known landmark nodes and computes coordinates from a high-dimensional Cartesian space for itself. Internet distance between two nodes is estimated as the Euclidean distance between their coordinates.

When a node S wishes to send a term list to nodes responsible for top terms in a document, it performs concurrent DHT lookups to locate all recipients. Each recipient directly replies to node S with its IP address and its GNP coordinates. After hearing from all recipients, node S locally builds a fully connected graph (a clique) with itself and recipients as vertices. It annotates each edge with estimated Internet distance, i.e., the Euclidean distance between the coordinates of the two nodes incident with the edge. In practice, other factors such as bandwidth and load could also be considered when assigning weights to edges. Finally, it runs a minimum spanning tree algorithm over the graph to find an efficient multicast tree.

The lookup results for recipients' IP and GNP coordinates are cached and reused for a certain amount of time. Stale information used before it times out will be detected when a node receives a term list that it should not be responsible for, which will trigger a recovery mechanism to find the correct recipient.

We use simulations to quantify the savings from overlay source multicast. Three data sets are used in our experiments. The first one is derived from a 1,000 node transit-stub graph generated by GT-ITM [6]. We use its default edge weight as the link latency. The second one is derived from NLANR's one-day RTT measurements among 131 sites scattered in the US [26]. After filtering out some unreachable sites, we are left with 105 fully connected sites and the latency between each pair of them. We use the median of one-day measurements as the end-to-end latency. The third one is an Internet Autonomous System (AS) snapshot taken by the Route Views Project [27] on April 28, 2003, with a total of 15,253 AS'es recorded. Since the latency between adjacent AS'es is unknown, we assign latency randomly between 8ms and 12ms. For the GT-ITM and AS data set, we randomly assign some user nodes to routers or AS'es. The end-to-end latency between two nodes is computed as the latency of the shortest path between them.

Figure 4 reports the performance of overlay source multicast using these data sets. In all experiments, we use the k-means clustering algorithm to select 15 center nodes as landmarks and then use GNP's technique to compute node coordinates from a 7-dimensional Cartesian space. For each data set, we randomly choose some nodes (shown on the X axis) as recipients to join the mul-

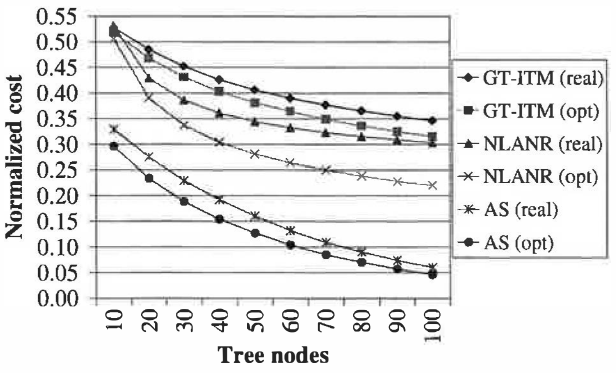


Figure 4: Cost of overlay source multicast normalized to that of using separate unicast to deliver data.

ticast tree. The cost of a multicast tree is the sum of the cost of all edges. The Y axis is the multicast tree's cost normalized to that of using separate unicast to deliver the term list to recipients. For each data set, the "opt" curve is the normalized cost of the optimal minimum spanning tree assuming that the real latency between each pair of nodes is known. The "real" curve (result of our scheme) is the normalized cost of the minimum spanning tree constructed using estimated Internet distance.

As can be seen from the figure, with 20 nodes in the tree, multicast reduces the communication cost by up to 72%. In all cases, the performance of the "real" curve approaches that of the "opt" curve, indicating that GNP estimates Internet distance with reasonable precision. The performance gap between the "opt" and "real" curve for the NLANR data set widens as the number of tree nodes increases. This is because GNP is not very accurate at estimating short distances. Some of NLANR's monitoring sites are very close to each other, particular at the east coast. This data set has many RTTs under 10ms. With more nodes added to the multicast tree, there is a bigger chance that some of them are very close to each other. Accordingly, GNP's estimation error increases and the resulting minimum spanning tree is less optimal.

5 Balancing Term List Distribution

One problem not addressed in previous studies on P2P keyword search [16, 20, 28, 39] is the balance of metadata distribution across nodes. Because popularity of terms varies dramatically, nodes responsible for popular terms will store much more data than others. A traditional approach to balance load in DHTs is *virtual server* [38], where a physical node functions as several virtual nodes to join the P2P network. As a result, the number of routing neighbors that a physical node monitors increases proportionally. More importantly, we find that virtual server is incapable of balancing load when the length of inverted lists varies dramatically.

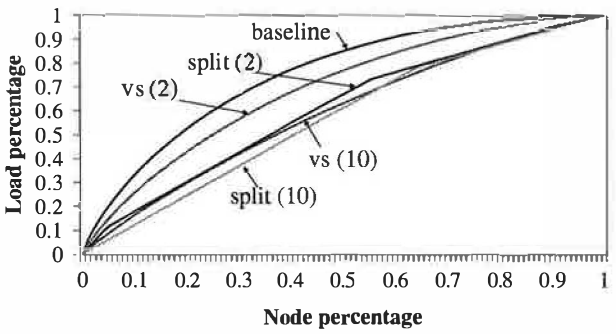


Figure 5: Comparison of load balancing techniques using equal-size objects. "baseline" is the basic Chord without virtual server. "vs (k)" is a Chord with k virtual nodes running on each physical node. "split (k)" is our technique where a new node performs k random lookups and splits the overloaded node.

Our solution is a combination of two techniques. First, we slightly modify Chord's node join protocol. A new node performs lookups for several random keys. Among nodes responsible for these keys, it chooses one that stores the largest amount of data and takes over some of its data. Second, each term is hashed into a key range rather than a single key. For an unpopular term, its key range is mapped to a single node. For a popular term, its key range may be partitioned among multiple nodes, which collectively store the inverted list for this term. Our experiments show that these two techniques can effectively balance the load even if the length of inverted lists varies dramatically. In addition, they avoid the extra maintenance overhead that virtual server introduces.

5.1 Node Join Protocol

In Chord, a new node performs a lookup for a random key K_r and splits the key range of the node n that is responsible for this key. Originally, node n is responsible for key range $[K_n^b, K_n^e]$. After the split, the new node and node n are responsible for key ranges $[K_n^b, K_r]$ and $[K_r + 1, K_n^e]$, respectively. With virtual server, the new node functions as several virtual nodes. Each virtual node executes this join protocol once.

Instead of splitting the key range of a random node, our technique seeks to split the key range of an overloaded node. In eSearch, a new node performs lookups for k random keys. Among nodes responsible for these keys, it chooses one that stores the largest amount of data to split. If nodes have heterogeneous storage capacity, it chooses the one that has the highest relative load to split.

Figure 5 compares the load balancing techniques. In this experiment, we distribute one million equal-size objects to a 10,000-node Chord. Nodes are sorted in decreasing order according to the number of objects they

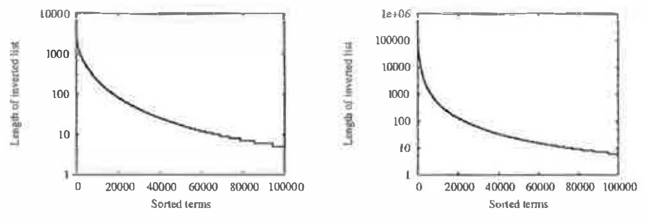

(a) (b)

Figure 6: Length distribution of the TREC corpus's inverted lists when documents are published under (a) top 20 terms, or (b) all terms.

store. The X-axis shows the percentage of the total number of nodes in the system for which the Y-axis gives the percentage of objects hosted by the corresponding nodes. "baseline" is the basic Chord without virtual server. "vs (k)" is a Chord with k virtual nodes running on each physical node. "split (k)" is our technique in which a new node performs k random lookups and splits the overloaded node. The closer the graph is to being linear, the more evenly distributed the load. This figure shows that our technique balances load better than virtual server, particularly when k is small. Note that this benefit is achieved without virtual server's maintenance overhead.

5.2 Distributing Load for a Single Term

As the popularity of terms varies dramatically, the length of inverted lists also does. Figure 6 plots the length distribution of TREC's inverted lists when documents are published under top 20 terms or all terms. In the following, we will simply refer to them as the "top 20 terms" load or the "all terms" load. Note that the Y axis is in log scale, and the numbers in Figure 6 (b) are larger than that in Figure 6 (a) by two orders of magnitude. The length distribution is skewed for both cases, particularly for the "all terms" load. Only publishing documents under the top 20 terms greatly reduces the length of long inverted lists corresponding to popular terms since they are actually not important in many documents they appear in.

Because of this variation in the length of inverted lists, the load balancing techniques in Section 5.1 cannot work well on their own. Our complementary technique is to hash each term t into a key range $[K_t^b, K_t^e]$ rather than a single key. The key range of an unpopular term is mapped to a single node whereas the key range of a popular term may be mapped to multiple nodes, which collectively store the inverted list of this term.

Chord uses 160-bit keys. We partition keys into two parts: 140 high-order bits and 20 low-order bits. Given a document D, we first identify its top terms. For each top term t, we generate a key K_t for it. The high-order bits

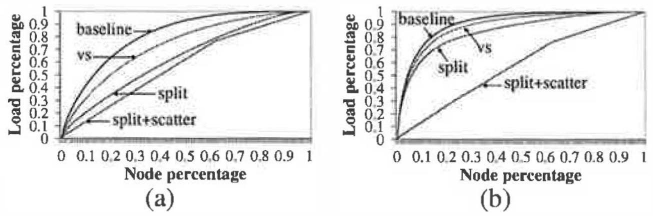

(a) (b)

Figure 7: Comparison of load balancing schemes using (a) the "top 20 terms" load, or (b) the "all terms" load.

K_t^h of key K_t are the high-order bits of the SHA-1 hashing [24] of the term's text. Let $K_t^b = 2^{20}K_t^h$. The low-order bits of K_t are generated randomly. We then store document D's term list on the node that is responsible for key K_t. As a result, the term lists for documents that have term t as one of their important terms will be stored in key range $[K_t^b,\ K_t^b + 2^{20} - 1]$. If this key range is partitioned among multiple nodes, then the inverted list for term t is partitioned among these nodes automatically.

The search process needs a change. A query containing term t is first routed to the node responsible for key K_t^b and then forwarded along Chord's ring until it reaches the node responsible for key K_t^b+2^{20}-1. All nodes in this range participate in processing this query since they hold part of the inverted list for term t.

The node join protocol is also changed slightly. When a new node arrives, it obtains a random document (through any means) and randomly chooses k terms that are not stopwords from the document. For each chosen term t, it uses the process described above to compute a key K_t for it. Among nodes responsible for these generated keys, it chooses one that stores the largest amount of data to split. The reason why we use random terms from a random document to generate the bootstrapping keys is to force the distribution of these keys to follow the inverted list distribution such that long inverted lists are split with a higher probability.

No other change to Chord is needed. Importantly, there is no specific node that maintains a list of nodes that store the inverted list of a given term, i.e., our technique is completely decentralized. Any node joining the partition of a term can fail independently. Node failure is handled by Chord's default protocol.

While the above describes the node join protocol in order to balance load in the presence of active joins, a similar protocol may be used by stable nodes to periodically redistribute load in the absence of joins. We do not implement or evaluate this optimization in this paper.

5.3 Experimental Results

Figure 7 compares the load balancing techniques. The "baseline", "vs', and "split" curves are the same as those

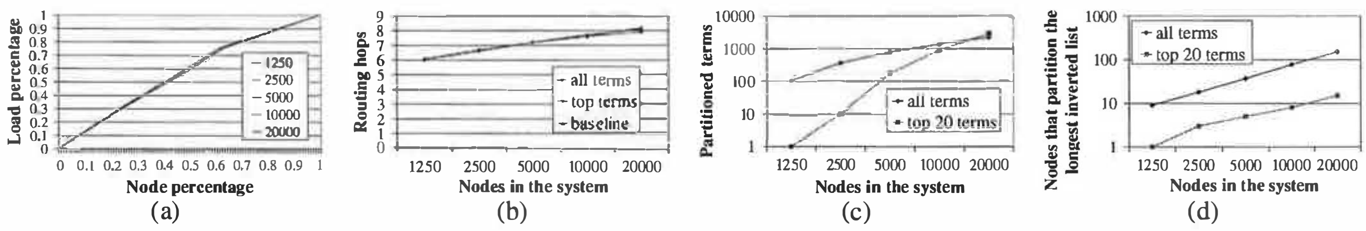


Figure 8: Scalability of our load balancing technique ("split+scatter"). (a) Metadata distribution. (b) Average routing hops. (c) Number of partitioned terms. (d) Number of nodes that partition the longest inverted list.

in Figure 5. "split+scatter" is our complete load balancing scheme, mapping a term into a key range and splitting overloaded nodes. For virtual server ("vs"), each physical node functions as 10 virtual nodes. For our techniques ("split' and "split+scatter"), a new node selects an overloaded node from 10 random nodes to split. We distribute the "top 20 terms" load and the "all terms" load to a 10,000-node Chord, respectively. Although the "all terms" load is not how eSearch actually works, we choose it to represent a scenario where the length distribution of eSearch's inverted lists becomes extremely skewed because, for instance, term lists for a gigantic number of documents are stored in eSearch.

In Figure 7, the load for the "baseline" is unbalanced, due to the large variation in length of inverted lists. For the "all terms" load, 1% of the nodes store 21.5% of the term lists. Virtual server improves the situation marginally—for the "all terms" load, 1% of the nodes still store 20.3% of the term lists. Splitting overloaded nodes ("split") performs better than virtual server ("vs") but only our complete technique ("split+scatter") is able to balance the load when the inverted lists are extremely skewed, owing to its ability to partition the inverted list of a single term among multiple nodes.

The rest of our experiments evaluate the scalability of "split+scatter", using the "all terms" load and varying the number of nodes in the system from 1,250 to 20,000. The load distribution is reported in Figure 8 (a). The curves overlap, indicating that "split+scatter" scales well with the system size. Figure 8 (b) shows the average routing hops in Chord with different configurations. "baseline" is the default Chord. The other two curves are "split+scatter" with different loads. All curves overlap, indicating that our modifications to Chord do not adversely affect Chord's routing performance.

Figure 8 (c) reports the number of terms whose inverted lists are stored on more than one node. Note that both the X axis and the Y axis are in log scale. As the number of nodes increases, the number of partitioned terms increases proportionally. The curve for the "top 20 terms" load grows faster. Because its inverted lists are not extremely skewed, more added nodes are devoted to partition unpartitioned terms, whereas in the "all terms" load added nodes are mainly used to repetitively partition terms with extremely long inverted lists. Figure 8 (d) reports the number of nodes that collectively host the longest inverted list. As the number of nodes increases, this inverted list is partitioned among more nodes proportionally. The partition in the "all terms" load is higher than that in the "top 20 terms" load because the inverted list in the "all terms" load is much longer.

Overall, our load balance technique scales well with the system size. It balances term list distribution well under different system sizes and does not affect Chord's routing performance. As the system size increases, long inverted lists are automatically partitioned among more nodes. Since it works for two quite different loads (particularly the extreme "all terms" load), we expect it to also scale well with the corpus size.

6 Analysis of System Resource Usage

In this section, we analyze eSearch's system resource usage when publishing metadata for a document or processing a query, and compare it with P2P systems based on global indexing (so-called *Global-P2P systems*) [16, 20, 28, 39]. We don't claim the default values used in the analysis are representative for all situations. Instead, we just want to give a flavor of eSearch's resource usage.

6.1 Publishing a Document

eSearch executes a two-phase protocol to publish a document. In the first phase, it uses DHT routing to locate the nodes responsible for top terms in the document and obtain their IP address and GNP coordinates. In the second phase, it uses overlay source multicast to deliver the document. The cost for the first phase could be avoided if the needed information has already been cached locally. We assume that this cache is disabled in the following analysis.

Data transmitted to locate the recipients are $B_l = n_t * h * m_l = 10,400$ bytes, where n_t=20 is the number of top terms, h=8 is the average number of routing hops in a 20,000-node Chord [1], m_l=65 is the size of the message (including 40-byte TCP/IP headers, 1-byte identifier that specifies the type of the message, 4-byte IP address of the data source, and a 20-byte DHT key). Data replied from the recipients are $B_r = n_t * m_r$=1,220 bytes, where m_r=61 is the size of the reply message (including 28-byte UDP/IP headers [2], 1-byte message identifier, 4-byte recipient IP address, and 28-byte GNP coordinates in a 7-dimensional Cartesian space). The total data transmitted in the first phase is the $B_l + B_r$=11,620 bytes.

In the second phase, there are two options for the content to be multicast to the recipients. The data source can build the term list for the document and multicast the term list. Alternatively, the data source can multicast the document itself, leaving it to the recipients to build the term list. The first method is more efficient in that a term list is smaller than a document, but the second method allows more flexible retrieval. With document text in hand, eSearch can search exact matches for quoted text, provide sentence context for matching terms, and support a "cached documents" feature similar to that in Google. We opt for multicasting the document itself.

Using statistics from the TREC corpus, we assume that the average document length is 3,778 bytes and it can be compressed to 1,350 bytes (a 2 to 4 text compression ratio is typical for bzip2). The UDP/IP headers, message identifier, and structural information of the multicast tree add 150 bytes to the multicast message, resulting in a 1,500-byte packet. Based on the simulation results in Section 4, we conservatively estimate that multicast can save bandwidth by 55%. Thus it consumes 20*1500*0.45=13,500 bytes bandwidth to multicast a document to 20 recipients. The total (phase one and phase two) cost for publishing a document is 11620+13500=25,120 bytes.

Next, we calculate the bandwidth consumption to distribute metadata for a document in a Global-P2P system. Although these systems [16, 20, 28, 39] did not propose using stopword removal, we add in this step. It significantly reduces the size of the metadata since as much as 50% of a document's content is stopwords. According to the TREC corpus, each document on average contains 153 unique terms after stemming and stopword removal. The metadata for a term in a document includes the term ID, the document ID, and a 1-byte attribute specifying, for instance, the frequency of the term in the document. (If this information needs more than one byte, approximating it with a 256-level value would provide sufficient precision.) The term ID is a DHT key of 20 bytes.

[1]The actual delay stretch in a proximity-aware overlay [18] may be smaller than the hop counts. This effect is discussed later.

[2]Throughout the analysis, we assume that communication between routing neighbors uses pre-established TCP connections whereas short communication between non-neighboring nodes use UDP.

We assume the document ID is 8 bytes, including the IP address of the node that stores the document, the port number through which to establish a connection with that node, and a 2-byte document number that differentiates documents on that node.

In a Global-P2P system, data transmitted to publish a document are $B = n_a * h * m = 80,784$ bytes, where n_a=153 is the number of terms in the document, h=8 is the routing hops in Chord, m=66 is the size of the message (including 40-byte TCP/IP headers, 1-byte message identifier, 16-byte term ID, 8-byte document ID, and 1-byte attribute). This bandwidth consumption is 3.2 times that of eSearch. This inefficiency is due to routing high-overhead small packets in the overlay network. Suppose the overhead of overlay routing can be reduced from h=8 to approximately h=2 by introducing proximity neighbor selection into Chord [18]. Then a Global-P2P system consumes 20,196 bytes bandwidth to publish a document whereas eSearch consumes 17,320 bytes bandwidth.

Global-P2P systems send a large number of small messages to publish a document. eSearch, in contrast, sends a small number of large messages (the whole document). If the inefficiency of processing a large number of small messages in routers and end hosts is counted in, savings in eSearch would be even more significant. Moreover, eSearch distributes the actual document, allowing more flexible retrieval.

6.2 Processing a Query

When a user intends to retrieve a small number of best matching documents, by default, query expansion is not used in eSearch (please refer to discussion in Section 3.3). The bandwidth cost to process a query is $B_q = n_q * h * m_q + n_q * m_d = 3,335$ bytes, where $n_q = 5$ is the number of nodes responsible for the query terms, $h = 8$ is the routing hops in the Chord, $m_q = 61$ is the size of the query message (including 40-byte TCP/IP headers, 1-byte message identifier, and 20-byte query text), and $m_d = 28 + 1 + 15 * (8 + 2) = 179$ is the size of the search results (including 28-byte UDP/IP headers, 1-byte message identifier, and 8-byte ID and 2-byte relevance score for 15 matching documents). Both query and publishing costs are independent of the size of the corpus and grow slowly (logarithmically) with the number of nodes in the system. In contrast, a local indexing system sends a query to every node in the system whereas the cost to process multi-term queries in a global indexing system grows with the size of the corpus.

Occasionally, the user is not satisfied with the search results and requests a feedback process to retrieve a larger number of documents using query expansion. The node that started the search first collects the metadata of feedback documents in order to select terms to be added into the query. The bandwidth cost is $B_c = n_f * m_f =$

$10,000$ bytes, where $n_f = 10$ is the number of feedback documents and $m_f = 1,000$ bytes is the cost to retrieve metadata for one feedback document. It then starts a second round of search using the expanded query. The analysis of bandwidth consumption is similar to that in the first round, $B_q = n_q*h*m_q+n_q*m_d = 315,510$ bytes, except that it searches 30 nodes ($n_q = 30$) and each node returns 1,000 documents ($m_d = 28+1+1000*(8+2) = 10,029$). In total, the feedback round consumes 325,510 bytes bandwidth, the majority of which is due to returning a large number of documents.

6.3 Storage Cost

The term list for a document is replicated about 20 times in eSearch, but its storage cost is not 20 times that of the Global-P2P systems, as explained below. To speed up query processing, each eSearch node locally builds inverted lists for documents that it holds term lists for. It also maintains a table that maps a document's 8-byte global ID into a 2-byte local ID. Although further compression is possible [43], we assume that 3 bytes are used for each (non-stopword) term in a document, 2 bytes for the local document ID, and 1 byte for an attribute (e.g., the frequency of the term in the document). The total cost to store metadata for a document in eSearch (including the cost for the mapping table) is 20*(8+2+153*(2+1))=9,380 bytes, assuming each document has 153 terms after stemming and stopword removal. Since storing full document text is an additional feature, we do not count it in the comparison.

Global-P2P systems need at least 9 bytes for a term in a document, 8 bytes for the global document ID and 1 byte for the attribute. The total storage cost for a document's metadata is 153*(8+1)=1,377 bytes. In these systems, information for terms in a document is distributed on different nodes. They cannot benefit from the technique that maps a document's long global ID into a short local ID since the size of one entry of the mapping table already exceeds the size of the information for a term and is not reused sufficiently to justify the cost.

The storage space consumed by eSearch is 6.8 times that of the Global-P2P systems. The benefit is its low search cost. According to Blake and Rodrigues [2], disk capacity has increased 160 times faster than the network bandwidth for an end user. Therefore, we believe trading modest disk space for communication and precision is a proper design choice for P2P systems. We also plan to adopt index compression [43] and pruning [7] to further reduce storage consumption.

7 Related Work

Compared with our P2P architecture, distributed IR systems such as GlOSS [17] uses a hierarchy of metadatabases to summarize contents of other databases. During a search, the summary is referenced to choose databases that may contain most relevant documents. We use semantic information produced by VSM to guide term list replication. Carmel et al. [7] used similar information to guide index pruning in a centralized site.

Below, we classify recently proposed P2P search systems into three categories according to the type of network in which they operate.

Search in Distributed Hash Table Systems

Global indexing based P2P keyword search systems built on top of DHTs are most relevant to eSearch. To answer multi-term queries, these systems must transmit inverted lists over the network to perform a join. Several techniques have been proposed to reduce this cost.

In KSS [16], the system precomputes and stores results for all possible queries consisting of up to a certain number of terms. The number of possible queries unfortunately grows exponentially with the number of terms in the vocabulary. Reynolds and Vahdat [28] adopted a technique developed in the database community to perform the join more efficiently. This technique transmits the Bloom filter of inverted lists instead of the inverted lists themselves. Suel et al. [39] adopted Fagin's algorithm to compute the top-k results without transmitting the entire inverted lists. This algorithm transmits the inverted lists incrementally and terminates early if sufficient results have already been obtained. Li et al. [20] suggested combining several techniques to reduce the cost of a distributed join, including caching, Bloom filter, document clustering, etc.

These approaches are orthogonal to our efforts in eSearch. Our hybrid indexing architecture intends to completely eliminate the cost for distributed join. A quantitative comparison of the search cost between them and eSearch would be an interesting subject of future work. Even with various optimizations, we still expect their cost to grow with the corpus size, perhaps at a rate slower than that of a basic global indexing system.

Search in Unstructured Peer-to-Peer Networks

Centralized indexing systems such as Napster suffer from a single point of failure and bottlenecks at the index server. Flooding-based techniques such as Gnutella send a query to every node in the system, consuming huge amounts of network bandwidth and CPU cycles. To reduce the search cost, heuristic-based approaches try to direct a search to only a fraction of the node population.

Rhea and Kubiatowicz [29] described a method in which each node uses Bloom filters to summarize its neighbors' content. A query is only forwarded to neighbors that may have relevant documents with a high probability. PlanetP [13] uses Bloom filters to summarize content on each node and floods the summary to the entire system. Crespo and Garcia-Molina [12] introduced the notion of Routing Indices that give a promising "direction" toward relevant documents.

Replication has also been explored to improve search efficiency. FastTrack [15] designates high-bandwidth nodes as super-nodes. Each super-node replicates the indices of several other nodes. Cohen et al. [11] found that setting the number of object replicas to the square root of the searching rate for an object minimizes the expected search size on successful queries.

Lv et al. [22] found that random walk and expanding-ring search are more efficient than flooding. Chawathe et al. [8] combined several techniques, including random walk, topology adaption, replication, and flow control, to improve Gnutella.

Search in Networks with Semantic Locality

Schwartz [35] described a method that organizes nodes with similar content into a group. A search starts with random walk but proceeds more deterministically once it hits in a group with matching content. SETS [1] arranges nodes into a topic-segmented overlay topology where links connect nodes with similar content. Motivated by research in data mining, Cohen et al. [10] used guide-rules to organize nodes satisfying certain predicates into an associative network. Sripanidkulchai et al. [37] extended an existing P2P network by linking a node to other nodes that satisfied previous queries.

Unlike the above systems, pSearch [40] is a P2P IR system that employs statistically derived conceptual indices instead of keywords for retrieval. pSearch uses latent semantic indexing (LSI) to guide content placement in a Content-Addressable Network (CAN) such that documents relevant to a query are likely be colocated on a small number of nodes. During a search, both pSearch and eSearch transmit a small amount of data and search a small number of nodes, but eSearch is relatively more efficient if compared quantitatively. In pSearch, the problem of efficiently deriving the conceptual representation for a large corpus is also very challenging. pSearch's infrastructure, however, supports content-based retrieval of multimedia data such as image or music files.

8 Conclusion

In this paper, we proposed a new architecture for P2P information retrieval, along with various optimization techniques to improve system efficiency and the quality of search results. We made the following contributions:

- Challenging conventional wisdom that uses either local or global indexing, we proposed hybrid indexing that employs selective term list replication to combine the benefits of local and global indexing while avoiding their limitations.
- We used semantic information provided by modern IR algorithms to guide the replication. The term list of a document is only replicated on nodes corresponding to important terms in the document.
- We adopted automatic query expansion in a P2P environment to alleviate precision degradation introduced by selective replication.
- We devised a novel overlay *source* multicast protocol that has very low protocol overhead in order to reduce term list dissemination cost.
- We introduced two techniques to balance term list distribution to a greater degree than is achievable using existing load balancing techniques while avoiding their maintenance overhead.

We have quantified the efficiency of eSearch (in terms of bandwidth consumption and storage cost) and the quality of its search results by experimenting with one of the largest benchmark corpora available in the public domain. Our results show that the combination of our proposed techniques results in a system that is scalable and efficient, and achieves search results as good as a centralized baseline that uses Okapi.

Our future work includes studying eSearch's sensitivity to the global statistics produced from samples, incorporating a P2P implementation [34] of Google's PageRank algorithm, and implementing index compression [43] and pruning [7] to reduce storage consumption. We also plan to experiment with a large HTML corpus crawled from the Web.

Acknowledgments

This work is related to the pSearch project [40] that Chunqiang Tang worked on during his internship at HP Labs in year 2002. We thank Mallik Mahalingam and Zhichen Xu for their contributions in the pSearch project; Gautam Altekar, Wenrui Zhao, the anonymous reviewers, and our shepherd Robert Morris for their valuable feedback; and Liudvikas Bukys and James Roche for their support in maintaining the experimental platform. This work was supported in part by NSF grants CCR-9988361, CCR-0219848, ECS-0225413, and EIA-0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; by the U.S. Dept. of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by equipment or financial grants from Compaq, IBM, Intel, and Sun.

References

[1] M. Bawa, G. S. Manku, and P. Raghavan. SETS: Search Enhanced by Topic Segmentation. In *SIGIR'03*, 2003.

[2] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *HotOS'03*, 2003.

[3] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *SIGMETRICS'00*, 2000.

[4] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[5] C. Buckley. Implementation of the SMART information retrieval system. Technical Report TR85-686, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1985. Source code available at ftp://ftp.cs.cornell.edu/pub/smart.

[6] K. Calvert, M. Doar, and E. W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 1997.

[7] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovicl, Y. S. Marrek, and A. Scoffer. Static Index Pruning for Information Retrieval Systems. In *SIGIR'01*, 2001.

[8] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM'03*, 2003.

[9] Y. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *SIGMETRICS'00*, 2000.

[10] E. Cohen, A. Fiat, and H. Kaplan. Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. In *INFOCOM'03*, 2003.

[11] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *SIGCOMM'02*, 2002.

[12] A. Crespo and H. García-Molina. Routing Indices for Peer-to-peer Systems. In *ICDCS'02*, 2002.

[13] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[14] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP'01*, 2001.

[15] FastTrack Peer-to-Peer technology company, 2001. http://www.fasttrack.nu.

[16] O. D. Gnawali. A Keyword Set Search System for Peer-to-Peer Networks. Master's thesis, Massachusetts Institute of Technology, 2002.

[17] L. Gravano, H. García-Molina, and A. Tomasic. GlOSS: text-source discovery over the Internet. *ACM Transactions on Database Systems*, 24(2), 1999.

[18] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *SIGCOMM'03*, 2003.

[19] The IRIS project. http://www.project-iris.net.

[20] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS'03*, 2003.

[21] X. Long and T. Suel. Optimized Query Execution in Large Search Engines with Global Page Ordering. In *VLDB'03*, 2003.

[22] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *ICS'02*, 2002.

[23] M. Mitra, A. Singhal, and C. Buckley. Improving Automatic Query Expansion. In *SIGIR'98*, 1998.

[24] National Institute of Standards and Technology. Secure Hash Standard, FIPS 180-1, 1995.

[25] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM'02*, 2002.

[26] NLANR. http://watt.nlanr.net/.

[27] Oregon Route Views Project. http://routeviews.org.

[28] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Middleware'03*, 2003.

[29] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *INFOCOM'02*, 2002.

[30] K. M. Risvik and R. Michelsen. Search Engines and Web Dynamics. *Computer Networks*, 39(3):289–302, 2002.

[31] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *TREC-3*, 1994.

[32] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP'01*, 2001.

[33] G. Salton, A. Wong, and C. Yang. A Vector Space Model for Information Retrieval. *Journal for the American Society for Information Retrieval*, 18(11):613–620, 1975.

[34] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. Distributed Pagerank for P2P Systems. In *the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[35] M. Schwartz. A Scalable, Non-Hierarchical Resource Discovery Mechanism Based on Probabilistic Protocols. Technical Report TR CU-CS-474-90, University of Colorado, 1990.

[36] A. Singhal. Modern Information Retrieval: A Brief Overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[37] K. Sripanidkulchai, B. Maggs, and H. Zhang. Enabling Efficient Content Location and Retrieval in Peer-to-Peer Systems by Exploiting Locality in Interests. *ACM SIGCOMM Computer Communication Review*, 32(1), 2001.

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, 2001.

[39] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval. In *WebDB'03*, 2003.

[40] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In *SIGCOMM'03*, 2003.

[41] Text Retrieval Conference (TREC). http://trec.nist.gov.

[42] A. Tomasic and H. Garcia-Molina. Query Processing and Inverted Indices in Shared-Nothing Document Information Retrieval Systems. *VLDB Journal*, 2(3):243–275, 1993.

[43] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, second edition, 1999.

Untangling the Web from DNS

Michael Walfish[a], Hari Balakrishnan[a], and Scott Shenker[b]
IRIS Project
[a]{mwalfish, hari}@csail.mit.edu, MIT Computer Science and AI Laboratory (CSAIL), Cambridge, MA
[b]shenker@icsi.berkeley.edu, International Computer Science Institute (ICSI), Berkeley, CA

Abstract

The Web relies on the Domain Name System (DNS) to resolve the hostname portion of URLs into IP addresses. This marriage-of-convenience enabled the Web's meteoric rise, but the resulting entanglement is now hindering both infrastructures—the Web is overly constrained by the limitations of DNS, and DNS is unduly burdened by the demands of the Web. There has been much commentary on this sad state-of-affairs, but dissolving the ill-fated union between DNS and the Web requires a new way to resolve Web references. To this end, this paper describes the design and implementation of *Semantic Free Referencing (SFR)*, a reference resolution infrastructure based on distributed hash tables (DHTs).

1 Introduction

DNS's original goal was practical and limited—allow users to refer to machines with convenient mnemonics [20, 21]—and it has performed this service admirably. However, with the advent of the Web and the resulting commercial value of DNS names, profit has replaced pragmatism as the dominant force shaping DNS. Legal wrangling over domain ownership is commonplace, and the institutional framework governing the naming system (*i.e.*, ICANN) is in disarray. Commercial pressures arising from its role in the Web have transformed DNS into a branding mechanism, a task for which it is ill-suited.

At a logical level, a linked, distributed system such as the Web requires a *Reference Resolution Service* (RRS) to map from *references* (our generic name for links or pointers) to actual locations. In the current Web, references are URLs with a hostname/pathname structure, and DNS serves as the RRS by mapping the hostname to an IP address where the target is stored. As the Web has matured, content replication and migration have become more important. However, the host-based nature of URLs—which ties references to specific hosts and hard-codes a path—makes content replication and movement hard.[1] Consequently, there have been many sensible calls, most notably in the URN literature [2, 5, 9, 19, 28, 29], to move the Web away from host-based URLs.

Since the Web has imposed the burden of branding on DNS, and DNS has restricted the flexibility of the Web, we believe that both systems would benefit if they were disentangled from each other. However, dissolving this mutually unhealthy union would require a new RRS for the Web. What should such an RRS look like? There has been extensive discussion about this topic, largely within the URN community but among many others as well. While we don't provide a comprehensive review of the commentary, the literature suggests the following two basic requirements for any such RRS (both of which DNS-based URLs do not satisfy):

Persistent object references: *A Web reference, like any abstraction used for indirection, should always be invariant, even when the referenced object moves or is replicated*. This principle has been central to the discussion about URNs. Reference persistence implies that references should not be tied to particular administrative domains or entities, as they are currently in DNS.[2]

Contention-free references: *Reference choice should be free of ownership disputes or other forms of legal interference*. Disputes over human-readable names are inevitable [22], but the reference resolution infrastructure is a poor place to resolve those disputes. Thus, as has been observed in the past [1, 10, 24, 29], references should be *inherently* human-unfriendly; in fact, we believe the infrastructure should enforce this property. Of course, users must be able to associate meaning to references, but the binding between human-friendly names and these opaque references should be done *outside* the referencing infrastructure. Such a separation would (a) free the RRS to focus only on technical concerns and (b) permit multiple, competing solutions to human-friendly naming, thereby allowing the resulting tussle [3] to play out through legal and other social channels.

[1]Because DNS names hosts, not Web objects, it is easy to move and replicate *hosts*. But DNS requires the sophisticated algorithms and substantial infrastructure of content distribution networks (CDNs) to achieve the same goals for individual Web *objects*.

[2]For instance, consider the Web page of someone who first created the page while at institution X but later moves to institution Y. If the reference record is controlled by the X domain (as it is with DNS) then maintaining persistence would require that X allow the author to update the record (if only to provide an HTTP redirect) for all time, even when the author is no longer affiliated with X. This expectation is impractical.

To these two well-accepted requirements, we add a third and less universally accepted design goal (which is similar in spirit to the goal articulated in [33]).[3]

General-purpose infrastructure: *The RRS should be designed to support a wide class of "link-based" applications.* The use of links, or pointers, to refer to objects or content on other machines is not unique to the Web; links are used in a variety of distributed systems for identifying objects and invoking remote code, for locating devices, and for other purposes when one wants to refer to objects by name, not location. The URN literature deals with this multiplicity by having context-specific resolvers [9]. However, since reference resolution is a hard problem that requires delicate design, we believe it should, if possible, be solved once and well.

How does one build a general-purpose RRS for persistent and contention-free references? In our work here we followed two key design principles:

Semantic-free namespace: We believe that the simplest way to achieve persistence and contention-free references is to use a namespace devoid of explicit semantics: a reference should neither embed information about the organization, administrative domain, or network provider it originated in or in which it is currently located, nor be human-friendly. We call such references *semantic-free*.

Minimal RRS interface and factored functionality: A general-purpose RRS should not impose unwanted semantics on applications, implying that the RRS should support a minimal interface limited to reference resolution. Therefore, all other functions required by applications—including mapping between human-friendly names and references—should be handled by auxiliary systems. We believe that the RRS's job is to *provide a platform* that allows for competition and flexibility in application-specific support and not to *solve directly* these higher-level problems.

We used these two design principles to develop both an RRS with **semantic-free references** (SFR) and a version of the Web that uses only SFR. The result is a system decomposition that differs from today's Web: whereas humans today rely on being able to read, and occasionally type, references (URLs), the Web-over-SFR handles user-level naming outside the reference resolution service by enabling a competitive market for *canonicalization services* that map human readable names to semantic-free tags. In the Web-over-SFR, search engines function as they do today, except they return links backed by semantic-free tags rather than by DNS-based URLs. Web browsers in turn use the SFR infrastructure to resolve these semantic-free tags to meta-data like IP addresses, ports, and pathnames that identify Web objects.

[3]The Globe project [32, 33] shares many of the same motivations as SFR, but, as we discuss in Section 7.2, the set of technical challenges addressed is rather different.

In addition to a different factoring, SFR enables new functionality for the Web, including: object-based migration wherein objects can move without requiring referring links to be updated or broken; flexible object replication wherein individual objects can be replicated without heavyweight machinery, administrative control over the hosts of the replicas, or hard-coding the administrative entity responsible for the meta-data; and content location services wherein individuals can provide reliable pointers to objects they did not contribute.

SFR is a "clean-sheet" design; not only does our design, in its pure form, require changes to all Web browsers, it also requires an infrastructure that currently does not exist. The Web-over-SFR is incrementally deployable via Web proxies, and a transition strategy exists, but we will not dwell on these methods. Our goal is rather to investigate, without regard to deployment issues, how one might best support the Web and other applications that require reference resolution. We hope that the lessons learned in this exercise will be useful, indirectly if not directly, in any future evolution of the Web and DNS.

2 SFR Challenges

SFR's advantages do not come without cost. Many of the desirable features of today's Web derive from DNS. As examples, DNS's hierarchical structure enforces URLs' uniqueness and provides fate sharing (a disconnected institution can still access local pages) while the human readability of DNS hostnames gives users some (perhaps misguided) confidence they have reached their desired data. Since SFR has abandoned both hierarchical structure and human readability, the SFR design must explicitly provide for the properties we have mentioned and others like them. Some of these challenges must be met by SFR itself, and some should be left to auxiliary systems supporting the Web-over-SFR. Addressing these challenges is the main focus of this paper. We now briefly discuss them and defer solutions to Sections 3 and 4.

2.1 SFR Infrastructure Challenges

Scalable resolution. Until recently, there was no way to scalably resolve references in a semantic-free namespace, which is largely why the URN literature chooses a partitioned set of resolvers [9]. However, the recently developed DHT technology [15] is designed to do exactly this: at their core, DHTs map an unstructured key from a flat namespace to a network location responsible for the key. But typical DHTs require $O(\log n)$ hops per lookup in an n-node system and would introduce intolerable latency. Thus, SFR must provide, on average, significantly faster lookups than the usual DHT performance bound.

Security and integrity. Any RRS must secure content providers' meta-data and enforce *reference integrity* by preventing two logically distinct objects from receiving the same reference. DNS guarantees these properties by relying on the administrator of each delegated namespace to protect meta-data and avoid local conflicts. A semantic-free namespace, however, has no natural administrative partitioning and thus protecting references—even under network partitions and malicious clients—is non-trivial.

Fate sharing. By delegating its namespace, DNS naturally offers fate sharing: if a domain (*e.g.*, `foo.edu`) becomes disconnected from the rest of the Internet, users in that domain can usually still access content served by that domain (*e.g.*, content at `x.foo.edu`), since the authoritative name server (*e.g.*, for `foo.edu`) is typically on their side of the partition. Since semantic-free names do not reflect objects' origin, SFR must be explicitly structured to provide fate sharing.[4]

Trust and financing. DNS has a very simple financing and trust model: organizations provide the *authoritative* server for their own domain, and DNS nodes need only serve requests *for* hosts or *from* users within the domain. Moreover, DNS requires only a small common infrastructure—the root servers—so all other expenses are incurred by the organization reaping the benefit (by allowing others to access their hosts). References in SFR are not tied to the content provider, so the "serve your own" trust and financing model does not apply.

2.2 Web-over-SFR Challenges

When the Web (or other applications with human interaction) runs over SFR, two important challenges arise: how users will find objects and how users can be sure that the content they see corresponds to the object they are seeking. *Rather than seek a single all-encompassing solution to these problems, we instead factor our system so that multiple, competing solutions can arise.*

Canonical names. There is good reason for the contention over DNS domains; they allow URLs to serve as *canonical names* that are memorable, human-readable, and easily transcribed. In contrast, the SFR approach provides opaque bit strings with none of these useful features. There is great benefit in simple and recognizable URLs, such as `http://www.cnn.com`. Thus, similar sets of canonical names that users can remember, understand, and transcribe must exist in the SFR framework.

Confidence. Humans browsing the Web are usually confident that URLs beginning with `www.nytimes.com` identify content published by *The New York Times* newspaper. While this reliance on the human semantics of a URL is hardly foolproof (as recent scams [4] have demonstrated), it does represent an important user need. SFR must clearly provide an alternative mechanism for giving users confidence in the content they are viewing.

`SFRTag:`	`0xf01212099abcab678ac345ba4d...`
`location:`	*`(ip, port),(DNS name, port), SFRTag`*
`oinfo:`	*`App-specific meta-data`*
`ttl:`	*`time-to-live: a caching hint`*

Figure 1: The `o-record`

3 SFR Design

Our proposed SFR system is a *shared infrastructure* that provides a single service: mapping from a semantic-free tag that references an object to meta-data associated with the object. Content providers insert an object's meta-data into the infrastructure and associate it with a tag. Consumers of the content submit these tags to the infrastructure and receive object meta-data in response. In this section, we focus on this single service and not on auxiliary questions, like how human-friendly names are mapped to tags.

3.1 Essentials

SFR uses a distributed hash table (DHT) to map semantic-free 160-bit strings, `SFRTags`, to `o-records` ("object records"). The `o-record`, shown in Figure 1, contains an object's location and other meta-data. The SFR infrastructure does not store objects, only their `o-records`. Our implementation uses Chord [31] as the underlying DHT routing protocol and DHash [8] to store the `o-records`, but the SFR architecture is modular and permits another DHT protocol to be substituted. In fact, SFR could use any system that supports scalable lookups on unstructured identifiers (such as the location service in the Globe system [32, 33]). For convenience, we will refer to DHTs as the fundamental resolving technology.

The `location` field is set by the application inserting the `o-record` and holds one or more values describing the location of the data corresponding to the `SFRTag`. Each `location` field entry is either an IP address and transport port pair, a domain name and transport port pair, or another `SFRTag` (that in turn resolves to another `o-record`). These latter two options permit additional degrees of indirection so that if many objects migrate together, they can all be updated by a single change to, respectively, DNS or SFR (the SFR option allows objects to move to different hostnames, while the DNS option enables changes to IP addresses for a fixed hostname). SFR's use of DNS to abstract the location of a *host* is not a contradiction; as we noted before, DNS is designed for exactly this function.

The resolving infrastructure imposes almost no constraints on applications since the structure, length, and

[4]We must address fate sharing because we insist on semantic-free references not because we use DHTs. The SkipNet DHT [12] provides fate sharing, but it encodes administrative domains into references.

content of the `oinfo` field are application-defined; *e.g.*, for the Web application, the field could hold the type of transport protocol (HTTP, FTP, HTTPS), a pathname on the server, etc. The SFR infrastructure does not look at this field. Finally, like DNS's TTL, the `ttl` field in the `o-record` is a caching hint instructing entities outside the infrastructure about how long to cache a given `o-record`. Because `SFRTags` are persistent references, the copy of the `o-record` in the infrastructure never expires and so `SFRTags` cannot be reassigned. As a result, if a content provider wishes to retire an `o-record` because the reference is no longer valid, the content provider empties the `o-record`.

The trust and economic model we envision for SFR is quite different from that of DNS because one cannot "serve your own" when the tags are semantic-free. So, instead of a DNS-like infrastructure comprised of "donated" machines dedicated to specific domains, we envision a more uniform infrastructure in which SFR nodes are trusted to serve all `o-records`. While there may be a startup period during which an "angel" (*e.g.*, NSF, European Union) funds the initial infrastructure (which may require, say, only 1,000 machines), once SFR becomes accepted as a viable service there could be competing commercial offerings. We believe that eventually several competing SFRs could peer with each other (exchanging updates) much like today's tier-1 ISPs, each holding mirror copies of all data. These peered SFRs would together form the global SFR infrastructure.

These SFR structures would be managed infrastructures with good connectivity (*we repeat: even though we are using DHTs, which are a so-called P2P technology, we are* **not** *relying on flaky personal machines connected via cable modems!*), so the SFR infrastructure machines would be relatively stable[5] and bandwidth between them relatively plentiful. Obviously the issue of the economics of such infrastructures is an open question, and our design thus relies on the shaky premise that competitive SFR infrastructures would arise; however, here we hope to convince the reader only that such infrastructures would indeed offer a better solution to the problems in today's Web and other linked services.

Before describing the rest of SFR's design, we emphasize that SFR's challenges derive from its semantic-free namespace, and it is this characteristic, rather than the particular choice of resolving substrate, that identifies SFR. One way to implement SFR may in fact be to use semantic-free DNS names. Indeed, one transition strategy is deploying various nameservers for a `.sfr` domain that would look up references in an SFR infrastructure. We do not view this as a contradiction: our objective is not to eliminate DNS but to change the way the Web uses DNS to resolve references. A system of semantic-free names built on DNS would face the same challenges as SFR and would require similar machinery, such as a way to do scalable resolution in an unpartitioned namespace.

[5]Nevertheless, in Section 6, we demonstrate that lookup performance remains acceptable under node failures.

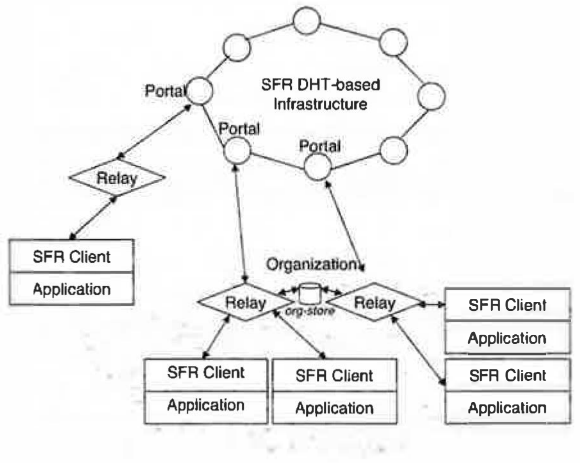


Figure 2: SFR components.

3.2 SFR Components

Figure 2 shows the components of the SFR system. At the core is the *SFR infrastructure*, a collection of managed nodes (of the kind we described above) that run *SFR server* software. This software runs on top of a DHT protocol and storage manager implemented at each node.

Applications store and retrieve `o-records` corresponding to `SFRTags` using the *SFR client* library. The client interacts with the SFR infrastructure using an *SFR relay*, a software module that intermediates between client requests for storing and retrieving `o-records` and the SFR infrastructure. The relay handles `o-record` caching and also ensures that clients can gain access to the `o-records` for content hosted by the local organization even when the organization is disconnected from the SFR infrastructure. The relay itself does not need to implement the DHT routing protocol or the storage manager; it connects to the SFR infrastructure at an *SFR portal*, which is simply a node in the infrastructure.

If SFR becomes widely deployed, client machines will need to discover a reachable SFR portal or relay. Clients today find out about available DNS servers via DHCP or via hard-coding; we envision identical techniques for SFR. Providing access to an SFR portal or relay would be one of the services offered by an ISP or large institution.

3.3 Security and Integrity

We now describe (a) how content providers who are clients of SFR may create unique, contention-free references without administrative namespace delegation and (b) how the infrastructure secures the content providers' meta-data. `SFRTags` and their associated `o-records` have these properties:

- The infrastructure ensures that SFRTags are the output of a hash function and thus have no human meaning.
- Content providers can create unique references without consulting a naming authority or any other entity.
- Only an o-record's creator, or someone who shares his private key, can update that o-record.
- Given a reference, the o-record is self-certifying.
- Content providers can update their public keys without invalidating references.
- The namespace is too massive for anyone to monopolize a significant chunk of it.

To achieve these properties, the SFR infrastructure first requires that an SFRTag is a secure, collision-resistant hash of the content provider's public key and an arbitrary salt. So when a content provider wishes to **create or update** a reference, it sends to its SFR portal (perhaps via an SFR relay), a request with all of the following elements:

- o-record, and SFRTag = hash(public key, salt);
- public key, salt, and version;
- signature (o-record, salt, version);

Before accepting this request, the responsible SFR infrastructure node checks that the SFRTag is the correct hash and that the signature is valid. The SFR infrastructure node then stores all of the data listed above. If the SFRTag was already in the infrastructure, the responsible node further checks that the request is signed with the current private key. (For clients who do not use public keys—and thus receive no protection—the SFR infrastructure also accepts references that are the hash of a client-chosen salt, only.) Because each reference is the output of a hash function, it is highly unlikely to have mnemonic or branding value, which in turn removes the need for a naming authority or other arbiter. In addition, SFR does not require, or use, a public key infrastructure.

On a **lookup**, a content consumer sends a request for the SFRTag to its SFR portal (perhaps via an SFR relay). If the tag is in the infrastructure, the responsible DHT node returns the corresponding o-record along with the auxiliary data mentioned above. Returning this data makes the o-record *self-certifying* [18]: *i.e.*, without resorting to a public key infrastructure, a retrieving client will be able to tell if a compromised node or malefactor in the middle of the network alters any of the data (since the reference is bound to the public key, and the signature binds the public key to the data). Moreover, SFR clients ensure that they are hearing from *bona fide* SFR infrastructure nodes by verifying the signatures on messages sent to them from the infrastructure nodes. We presume that when a client receives the address of an SFR node (*e.g.*, by DHCP, as with DNS servers), the notification also includes that SFR node's public key.

Public key updates do not invalidate the reference since the SFR infrastructure requires only that the relationship between the SFRTag and the public key is satisfied when the tag is *first* inserted. After that, the SFR infrastructure ensures that updates to the public key or to the content have been signed with the existing private key. To maintain the self-certifying property, the infrastructure must store the content provider's signed request to update its public key and must also return these signed requests in response to lookups.[6] To guard against replay attacks, SFR adopts DHash's approach [8]: SFR clients increment, and sign, a version number each time they update their o-record, and the infrastructure accepts updates only with increasing version numbers.[7]

The sheer size of the SFR namespace prevents anyone from monopolizing a significant portion. Protecting individual SFR nodes or the DHT as a whole against loading is a different matter, however. Our intent is that these attacks will be addressed by management tools to prevent content providers (where a content provider is defined by its public key) from using too many resources.

3.4 Latency

SFR uses three kinds of TTL-based caching to reduce latency and balance load among the infrastructure nodes. First, each relay caches o-records (and the auxiliary data like public keys and signatures), sharing that cache among the clients that use that relay. Because the use of a relay is optional, SFR clients also cache o-records.

Second, each DHT node in the infrastructure keeps a *location cache* of identifier-to-IP mappings for nodes it has recently heard about. This reduces the number of hops in certain DHT routing schemes that require $O(\log n)$ hops in an n-node system. In Section 6 we present simulation results showing that location caching can lower the number of hops to two or three in over 99% of lookups. "One-hop" DHT routing schemes [11] are another way to lower the number of hops.

Third, SFR infrastructure nodes also cache o-records, which helps balance load and ease hot-spots corresponding to highly popular o-records: such o-records will quickly be cached by the portals and so should not stress the infrastructure. In addition to each portal's caching the o-record retrieved on behalf of a relay or client, our design also permits nodes on the DHT lookup path to cache o-records, thereby proactively populating their cache.[8]

[6]Without additional infrastructure, the loss of a key could be catastrophic, but one could imagine auxiliary services that would serve as trusted and secure repositories of such keys. However, if a key is compromised, the situation is more dire; we think the only way the original owner can prevent the takeover of his content is to *break* all current tags (*i.e.*, render them unusable by anyone, adversary and victim alike).

[7]While we don't discuss replication explicitly here, DHTs need replication to provide reliability. Thus, retrieving clients may need to download from several locations to ensure they have the latest version number.

[8]In general, improving the performance of DHT-based systems is

3.5 Fate Sharing and Scoping

In the following, we define an "organization" as a set of machines behind an access link. If an organization corresponds to a single DNS domain, and if the organization's DNS servers are also behind the access link, then, when the link fails, hosts in the organization can continue to reach data within the organization. As described so far, SFR does not provide such organizational fate sharing because an organization's `o-records` are not explicitly associated with, or stored within, the organization.

However, SFR can ensure that clients in the same organization as the *creator* of an object can access the object when an access link fails, thereby replacing domain-based fate sharing with what we call *write-locality*-based fate sharing. The enabling mechanism is a shared *org-store* holding copies of the `o-records` created or modified within the organization. Each time a new `o-record` is created or modified via one of the relay nodes in the organization, the relay stores a copy in the org-store and arranges for it to be stored in the SFR infrastructure.

When retrieving, the relay first checks its internal `o-record` cache. If the `o-record` is in the local cache and the TTL is still valid, the relay returns the `o-record` to the client. Otherwise, the relay contacts its portal to initiate a lookup of the `SFRTag` in the SFR infrastructure. At the same time, the relay contacts the org-store, which returns the `o-record` corresponding to the tag if one exists, *disregarding* any TTL value set in the `o-record`. If the relay does not hear from the SFR infrastructure, it times out and infers that it cannot access any of the persistent copies in the infrastructure. It returns to the client the `o-record` returned by the org-store.

The reason the relay does not directly send the version from the org-store *before* waiting for a response from the SFR infrastructure is that another content provider, that shares the same private key but is located in *another* organization, may have updated the `o-record`. For this reason, whenever the relay retrieves an `o-record` from the infrastructure, it also sends a copy to the org-store so that the versions in the org-store and the infrastructure can be reconciled if necessary. The version number in the `o-record`, incremented on each update, and a UTC timestamp set by the writer indicating the last update time, facilitate this reconciliation.

Updating an `o-record` via a relay within the organization also requires the update to be sent *both* to the infrastructure and to the org-store. If the relay finds that the infrastructure store request does not succeed because of lack of connectivity, it asks the org-store to reconcile the `SFRTag` whenever the organization is reconnected. Updating an `o-record` that was originally created in a different organization does not immediately update the org-store in the creating organization; that update happens when the `SFRTag` is looked up via a relay in the original organization. This level of inconsistency is unavoidable without out-of-band synchronization.

The foregoing scheme improves availability for disconnected organizations but does not ensure that infrastructure nodes hold up-to-date versions of `o-records`. If an organization remains internally connected, the semantics of this write-locality-based cache are:

- For `o-records` created in an organization and never updated from outside the organization, clients within the organization always get the most recent version.
- For `o-records` updated by more than one organization, a client within the currently disconnected organization *may* receive an older version that is no older, and is possibly newer, than the last version written from within the organization.
- When connectivity between the SFR infrastructure and an organization is restored, (1) all subsequent retrievals from within the organization return the `o-record` with the highest version number, and (2) all subsequent retrievals from outside the organization return the latest version after it has been reconciled.
- Reconciliation of `o-records` uses either the later timestamp (which works reasonably well assuming loose clock synchronization between writers and preserves the same semantics as when multiple writers update an `o-record` while connected to the SFR infrastructure), or an out-of-band mechanism (*e.g.*, by discarding one of them, perhaps with human involvement). We believe this approach is reasonable because conflicting updates are likely to be rare and suggest the absence of higher-level human coordination.
- Unlike with DNS, clients in different organizations on the same side of a network partition are not guaranteed to be able to access the other organization's meta-data.

Scoping arises naturally in the org-store framework: if clients within the organization wish to limit their meta-data to the organization, the relay simply stores the `o-record` in the org-store only.

4 The Web-over-SFR

In today's Web, references (*i.e.*, URLs) encode the administrative entity (*i.e.*, the domain) responsible for an object's meta-data. Thus, if an object changes domains, hyperlinks to the object are almost guaranteed to break, and a human browsing the Web might get a "notify the referrer" message. Since references should not have to change

an active area of research, and we expect to use solutions from ongoing work in the community on data replication, load balance, denial-of-service defense, fault tolerance, and protection against compromised nodes. While solutions to these problems are not all currently at hand, we are optimistic that there is no fundamental obstacle to basing SFR on DHTs, and so we focus on the many SFR-specific problems.

```
SFRTag:   0xf01212099abcd3848123ab38121
(ip_addr1, port1, proto1, path1),
(DNS name, port2, proto2, path2), ...
```

Figure 3: Logical view of the `o-record` for the Web-over-SFR. The `proto` field specifies the access protocol (*e.g.*, HTTP, HTTPS, FTP). The `path` field is the local pathname on the server and identifies the referenced object to the server.

when objects move, we attempt, in the Web-over-SFR, to provide a set of references that cleanly permit object migration and replication.

As we have already noted, the current Web supports object migration only if the original *domain* (which may no longer have any connection with the content creator) issues HTTP redirects for objects it no longer hosts. In the Web-over-SFR, in contrast, *all of the information about how to reach a particular Web object*—the IP address and port of the Web server and the pathname on the Web server—is abstracted by the `SFRTag`. Content creators (*e.g.*, individuals, organizations, research groups) insert this reachability information into an `o-record` and store the `o-record` in the SFR infrastructure. To take advantage of these persistent references, Web authors embed hyperlinks like:

`sfr://f012120.../optional_path`

where `f012120...` is an `SFRTag` resolving to a set of tuples identifying the object, as shown in Figure 3.

To retrieve objects using this kind of URL, the Web browser uses the SFR client to fetch the meta-data and construct an HTTP request. The path in the HTTP request is the concatenation of a path from the `oinfo` field with the `optional_path` from the original URL. This design preserves HTTP's semantics. The SFR infrastructure is invisible to Web servers, which continue to receive HTTP GET requests with server-specific paths. The `optional_path` permits flexibility, as we describe below, and it also permits dynamic content because embedded links can have paths with application-specific semantics. Without the `optional_path`, content providers, Web clients, and Web servers would need to involve the SFR infrastructure to construct unique URLs. That these SFR-based URLs contain semantics illustrates that SFR allows applications to define their own semantics while still using a semantic-free referencing infrastructure.[9]

4.1 Benefits of Web-over-SFR

Resilient linking. The SFR approach permits a general migration solution: if a piece of content, currently referenced by an `SFRTag`, moves to another Web server at a different path, the content provider need only change the `location` and `oinfo` fields in the `o-record` in order to permit the correct reference resolution to occur for Web clients. Web pages linking to the object continue to maintain the same references.

This approach is flexible about how much the reference functions as an abstraction. An `SFRTag` can refer to a machine (so the `optional_path` is the same as it is with today's URLs), to a file (so the `SFRTag` abstracts the entire URL and the `optional_path` is empty), or to a directory structure (so the `SFRTag` abstracts the entire URL up until the root of the directory, and the `optional_path` is everything underneath the directory). For example, a researcher might have a large collection of publications in one directory and wish to abstract only the collection's location. In this case, the `SFRTag` would abstract the IP address or domain name of the Web server as well as the path on the server up until the document collection. The publications would be differentiated by their file names. So the SFR URLs could be:

```
sfr://fbcd123/pub1.ps
sfr://fbcd123/pub2.ps
```

If the researcher's affiliation then changes, he or she alters the `o-record` corresponding to `fbcd123` and inserts the new Web server and new path on the server. A referring Web page embedding `sfr://fbcd123/pub1.ps` can safely be ignorant of the move.

If a particular object separates from a directory that had been abstracted by an `SFRTag`, then, under the design as so far explained, existing references would break. Our solution for this case adds a level of indirection: the content owner would update the `o-record` to point to a new location that would maintain a map of old pathnames to new location/pathname values. Although this solution can implement HTTP redirection (if the new location were the same as the old server location), our solution does not mandate this approach. In Section 7.1, we discuss a more elegant, but more demanding, solution that does not require this level of indirection.

The main reason the solution we have described above is more powerful than today's DNS-based RRS for achieving resilient linking is that, unlike DNS, SFR is able to resolve both the tag and the pathname before any HTTP messages are sent to the Web server. Achieving similar behavior today would either require a prescient content provider to have a domain name for each potentially movable piece of content beforehand, or rely on HTTP redirection; the former is impractical and a management challenge, whereas the latter is hard to ensure when one moves between organizations.

Flexible object replication. SFR provides a natural solution for replicating Web objects: in response to a request for an `SFRTag`, the infrastructure can return a num-

[9]Note that DNS could certainly be enhanced with a record type that abstracted individual Web objects, instead of hosts, but as we explain in Section 7.2, such a system would either inherit the problems we have identified with today's use of DNS or else look very much like SFR.

ber of different logical locations and paths. This property might seem inconsequential, but consider how hard it would be under the current Web to replicate a given object in two places *without creating mirror machines containing* exactly *the same content*. To do so would require (a) creating separate DNS names for each *object* being replicated, (b) using virtual hosting so that the two Web servers were configured to recognize each per-object DNS name and (c) configuring DNS entries to refer to both Web servers. Using a domain like `www.personalname.org` would not work since that forces all objects in the domain to be resolved by the same administrative entity forever, making it impossible, *e.g.*, for an individual object like `www.personalname.org/photos` to migrate later without breaking existing, referring hyperlinks. The Web-over-SFR solution is much simpler and would allow several collaborators to replicate each other's content quite easily, yielding a *grass-roots* replication service.

In the case of *massive replication*, namely when it would be absurd or inappropriate to return to the client all of the locations of all replicas, we expect that the SFR infrastructure would direct clients to external services, such as a replication server that would direct requests to the appropriate replica using information from the requester's IP address and other hints. We discuss alternatives to this decision in Section 7.1.

Reliable pointer services. Because SFR permits anyone to insert `o-records` into the infrastructure, third parties can become known as good indirectors—they can create and expose `SFRTags` that always resolve to particular sites or objects, and it would be their responsibility to track the object's movements. Referring Web pages could embed the `SFRTags` established by the indirectors, and then these providers of reliable pointers might have an incentive to make the location service work. For example, a service might provide a pointer to "This year's tax forms"; no matter what year it is, you can access the necessary tax forms by following the `SFRTag`.

4.2 Human-Usability Challenges

The challenges that arise under the SFR framework but which are not explicitly solved by the infrastructure are related to *user-level names*, our term for the ways that humans identify content, such as search queries, typed-in URLs, AOL keywords, hyperlinks in documents, saved bookmarks, and URLs sent in e-mail. DNS-based Web URLs conflate the *reference*, a low-level tag resolved by the RRS, and the *user-level name*, which allows people to find what they are looking for. SFR, in contrast, separates the two functions, focusing on reference resolution and exposing an interface that permits many user-level naming solutions to co-exist.

Canonical names. A natural question is how humans will retrieve content if references are human-unfriendly. The answer is, first, that humans mostly do not depend on typed-in URLs today. All other current user-level naming methods work perfectly under SFR: search queries, for example, return candidate SFR URLs instead of DNS-based URLs. Individuals could also e-mail SFR URLs to each other. (Users are already used to dealing with human-unfriendly URLs this way: links sent in e-mail from `amazon.com`, for example, are essentially a domain name plus a semantic-free string.)

Second, when users do type URLs, they use DNS as a *canonicalization service*: a well-known mapping from human-readable names to Web objects. To permit equivalent functionality under SFR, DNS need not be the canonicalization source. Moreover, it might be desirable if several mapping services existed. We believe that if SFR becomes popular then Web service providers with appropriate expertise would compete to provide such services. Two obvious models already exist—AOL keywords and the paper yellow pages—and we can imagine a wide spectrum of services that map user-level names to a particular `SFRTag` or set of `SFRTags`.

We observe that to the extent DNS provides canonical handles today, it does so mainly for Web *sites* and seldom for individual Web *objects*. Since references in SFR can be as coarse as per-site or as granular as per-object, any canonicalization service for SFR would naturally be able to name entire sites and individual objects, ultimately yielding a more complete canonicalization function than DNS.

Of course, under SFR, the problem of bootstrapping exists, namely how users get pointers to directory services. A number of possibilities exist, including pointers shipped with browsers, links sent in e-mail from friends, applications on the local host that populate a local database of useful sites, and network administrators or ISPs dynamically providing pointers to useful canonicalization services with DHCP. DNS names themselves could provide one canonicalization service, *e.g.*, `www.foo.com` would map to an `SFRTag` for the home page for Foo, Inc. This is similar to the scheme presented by Ballintijn *et al.* [2]. Using DNS in this way is not a contradiction: we are not using DNS for *referencing* but rather as a user-level naming service that competes with any number of other such services.

Confidence. Although human users of the Web usually have confidence in Web content because of the associated DNS name, we note that this level of confidence is actually quite weak. It depends entirely on whether the "correct" company owns a given domain name, and it is easy to create spoof sites that give users misplaced confidence in content. Nevertheless, domain names convey meaning and help users validate URLs before visiting them (*e.g.*, when selecting among search results). Hence, we anticipate that search engines (and others) would hide SFR URLs and give humans confidence in new ways.

Rather than give lengthy detail on ways humans can have well-placed confidence in content under SFR, we outline just one (though imagining others is not hard): hyperlinks on Web pages optionally embed a `taginfo` object alongside the `SFRTag`. This object contains cryptographic statements of the form "Entity E says that this tag is CNN", where E is a Web service provider that users trust. Users' browsers would inform them about who is certifying the link. We note that this scheme is implemented entirely above the SFR layer and that it is an application function, leaving SFR performing only reference resolution. With such a scheme, content authentication could be granular—it could occur at the level of individual objects, rather than at the level of administrative domains (as certificates issued by certificate authorities do today, for example).

Rather than "hard-coding" one approach to canonicalization and confidence, we believe the infrastructure itself should permit multiple schemes to co-exist. Separating the functions of user-level naming and reference resolution will certainly not solve the intractable problem of humans fighting over names. But it will move these tussles [3] to an arena in which multiple services can compete but in which none of these competing services is part of the core reference resolution infrastructure.

4.3 Pragmatics

So far we have focused on the fundamental problems faced by SFR. However, there are more pragmatic concerns that would have to be addressed before the Web-over-SFR could be viable. We don't believe they represent insurmountable difficulties, but they will require new tools. We now mention a few of these issues.

Local references. Currently, with DNS-based URLs, if a hyperlink in a Web page points to another page in the same administrative domain, there are two possibilities: either the hyperlink will be local (*e.g.*, `<A HREF=/imgs/dog.gif>`) or the hyperlink will reference the current domain (*e.g.*, `<A HREF=http://mysite.org/imgs/dog.gif>`). The first case does not involve any lookups, so it would continue to work in an SFR-based Web (although, because references can abstract any portion of the path component, only absolute links will work, not relative ones). In the second case, under DNS, the client need not do another lookup because it would have the address information for `mysite.org` cached.

Under SFR, however, if the content provider used another `SFRTag` to refer somewhere on the same site (*e.g.*, `<A HREF=sfr://ab12126/dog.gif>`), the client would have to do a separate SFR lookup, incurring additional latency. Our (currently unimplemented) solution is to allow the content provider to insert hints next to *local* URLs. These hints would indicate that the reference is local and would also contain a pathname.

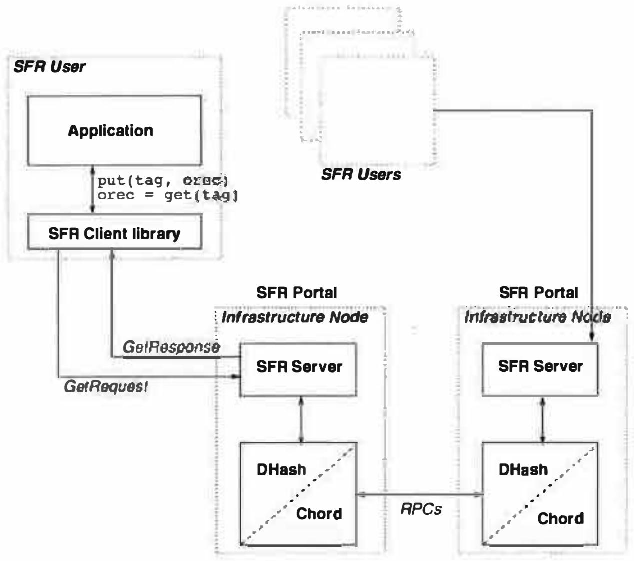


Figure 4: SFR implementation.

Optimizations. As described earlier, `o-records`' `location` fields may contain DNS names, possibly introducing extra latency (since clients would have to do two sets of lookups, one for SFR and one for DNS). If reducing latency were paramount for the `o-record` owner, however, the owner might avoid this layer of indirection and instead rely on an external system that directly updates the IP addresses in batches of `o-records`.

Tools. We believe that any realistic deployment of SFR would necessarily be accompanied by new editing tools for content providers that either hide the actual references or else make them easier to work with. Although questions about how to build these tools are worthwhile, they are outside the scope of this paper.

5 Implementation

5.1 SFR Implementation

SFR portal nodes run a slightly modified version of MIT's DHash/Chord [8, 31] along with a separate SFR server module that uses DHash's API to store and retrieve `o-records` in response to client requests. Client applications interact with SFR by linking to the SFR client library, which communicates with a nearby SFR portal via a simple request/response protocol, as pictured in Figure 4. We have not yet implemented the SFR relay. The SFR client library exposes `put()` and `get()` methods to applications for storing and retrieving `o-records`. Both the SFR server and SFR client cache `o-records` according to the TTL field.

The SFR server has several purposes: it abstracts the underlying DHT for applications that use SFR; it exposes a narrow interface (so that SFR clients need not conform to the wider interfaces that DHTs sometimes require); and it serves as a marshal for client requests, allowing SFR to control its clients' interaction with the DHT and allowing administrators to extend the SFR server to implement other security and access control functions.

To achieve reference integrity through randomness, we modified DHash to enforce the relationship described in Section 3.3 between the reference, a salt and a public key. Like DHash, the SFR server and SFR client are written in C++, use the SFS toolkit [17] for asynchronous programming and cryptographic operations, and run on FreeBSD and Linux. Because of the simple network protocol between the SFR client and server, we anticipate that writing SFR clients in other languages and on other platforms will not be difficult.

The protocol has four messages: `GetRequest`, `GetResponse`, `PutRequest`, and `PutResponse`. The messages' contents (including items like the salt, the `o-record`, and the public key) are sent using type-length-value (TLV) encoding; both the client and server sign their messages. Applications must supply the public key of the SFR portal to the SFR client library (the converse is not necessary because the SFR portal does not need to identify its clients, except by the public key, which the client supplies). The implementation currently assumes a stream-oriented connection (which is certainly not optimal for performance), but it would not require much effort to move to an unreliable service like UDP.

5.2 Web-over-SFR Implementation

SFR clients are not yet embedded in Web browsers and so to prototype the Web-over-SFR, we use a Web proxy that simulates how a Web browser would interact with SFR if SFR were ubiquitous. The proxy is written in C++ and uses the SFS toolkit and SFR client library. The proxy's basic operation is translating URLs submitted by clients into SFR URLs. The proxy serves several functions: (1) it allows end-users to experience the latency associated with SFR as compared to DNS; (2) it allows us to dynamically populate SFR with the `o-record`s that would exist if the whole Web used SFR; and (3) it allows us to test the usability of semantic-free URLs.

In its usual mode, the proxy addresses (1) and (2). When a client browser requests a traditional URL, the proxy translates it into an SFR lookup by first hashing the URL and using that hash as the salt, and then hashing this salt together with the proxy's public key, thereby creating an `SFRTag` (as described in Section 3.3). The proxy then uses the SFR client to retrieve meta-data for this `SFRTag`. If the lookup is successful, the proxy uses the IP, port, and pathname information in the returned `o-record` to contact the actual Web server and then begins returning content to the client, thereby incurring the latency associated with an SFR lookup. If the lookup is unsuccessful, the proxy, besides returning content to the client, populates the SFR infrastructure on demand by constructing an appropriate `o-record` (based on a DNS lookup) and inserting it into the infrastructure.

This `o-record` contains a list of (IP address, port) pairs as well as a corresponding list of paths, a timestamp, and the TTL from DNS (different from the `o-record`'s TTL field, discussed in Section 3.1). The proxy stores these latter two items to obey DNS's semantics: if the proxy does an SFR lookup and the TTL has expired, the proxy executes another DNS request and inserts the updated `o-record` into the infrastructure.

The SFR Web proxy also directly accepts URLs of the form `http://0123aa.../optional_path` and treats the `0123aa` portion as an `SFRTag` (as we described in Section 4). Given Web pages with this type of SFR URL, we can test SFR's usability. In the future, we plan to have the proxy also rewrite traditional URLs in the Web pages that it returns to clients to make these URLs semantic-free, thereby permitting convenient usability tests.

6 Evaluation

We analyze SFR's performance using a combination of real-world data and simulation.

6.1 Latency Data

We deployed SFR nodes running DHash/Chord and the SFR portal software on the PlanetLab testbed [26]. The Chord ring uses approximately 130 physical hosts and 390 virtual nodes [31]. We also deployed our Web proxy at three different PlanetLab locations, and seven people (including the authors) used this proxy for days at a time over a one month period. When the proxy receives a URL, it creates two `SFRTags`—one corresponding to the hostname portion of the URL and the other corresponding to the entire URL—and then submits both to the embedded SFR client (which in turn contacts the SFR portal running on the local host). In order to permit a fair experiential comparison with DNS, the proxy returns content to the user as soon as the SFR client returns the `o-record` corresponding to the hostname digest. The SFR client cache (which obeys DNS TTLs in our implementation) is then equivalent to a DNS cache. If SFR were actually deployed, the number of `SFRTags` would be in between the number of hostnames and the number of distinct URLs on the Web: many `SFRTags` will certainly refer to directories under Web sites but not to individual Web pages.

Figure 5 compares the CDF of SFR's latency (as measured by the SFR portals) to a dataset for DNS that depicts a CDF of latency, as measured by a resolver at MIT. Only the SFR lookups that resulted in a Chord lookup are rep-

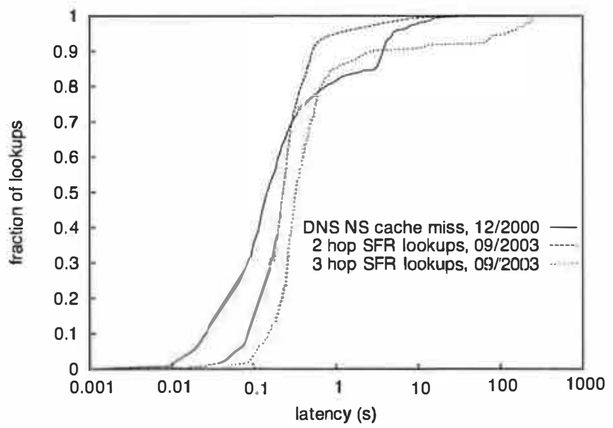


Figure 5: CDFs of SFR latency and DNS dataset

resented; the rest are satisfied via the SFR client's or the SFR portal's `o-record` cache. We use lookups from an eight-day period in September 2003; the depicted period occurred during the one month period mentioned above and after a bug fix that slightly improved latency. Because of the limited size of the PlanetLab Chord ring, aggressive caching of other virtual nodes' locations, and sharing of these caches among virtual nodes, 98% of the almost 15,000 lookups resolved in two Chord hops (the usual minimum); the rest required three hops. We show in simulation below that even in a large Chord ring, aggressive location caching results in two or three hops per lookup.

The DNS data comes from the work by Jung *et al.* [16] and depicts the end-to-end latency experienced by a resolver at MIT when NS record cache misses occurred. We do not incorporate A-record cache misses because doing so would unfairly count the many small requests for low-TTL A-records that are directed to CDNs, which move the name servers for popular content close to the client in many cases. We anticipate that if SFR were widely deployed, CDNs running over it would be able to implement similar optimizations. The DNS data is three years old and was collected at a single institution; hence, this comparison is meant to be suggestive, only, and not conclusive.

The feedback from our users is that perceived latency was generally indistinguishable from DNS, and Figure 5 supports this claim, suggesting that SFR's latency in the common case (two hops) is reasonably close to DNS's.

6.2 Simulation

We have just seen that on a testbed shared by hundreds of researchers, two and even three hop Chord lookups yield reasonable latencies. We now wish to confirm with simulations that—despite the $O(\log n)$ theoretical bound for number of lookups—two and three hop lookups will, in fact, be the norm when the hosts implementing the DHT do aggressive location caching.

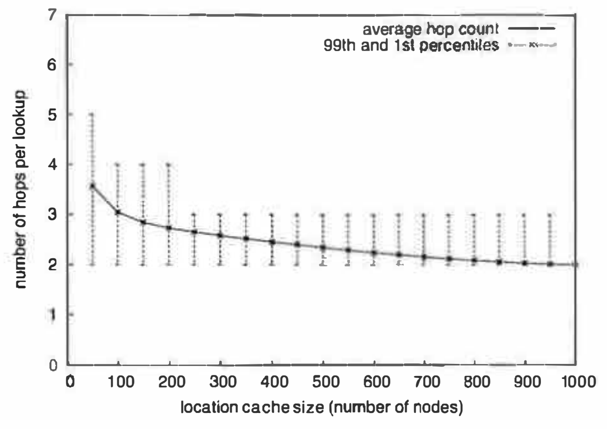
We used a modified version of the Chord simulator described by Stoica *et al.* [31] to gather trace-driven results. In the simulator, nodes add any node with which they communicate to the location cache. Eviction proceeds LRU, though a node's fingers will never be ejected. Because of Chord's routing, aggressive location caching causes nodes to accumulate relatively more information about nodes nearby in ID space.

To drive our simulations, we used two days of NLANR cache trace data [14], aggregating the separate caches' logs. Each URL in the aggregated trace causes a simulated SFR lookup of the URL's hash. (Hashing hostnames produced slightly better results, so we conservatively present the former.) To "warm up" the location cache, we ran the simulator on a day's worth of NLANR requests and then tabulated hop counts for the next 10^6 requests.

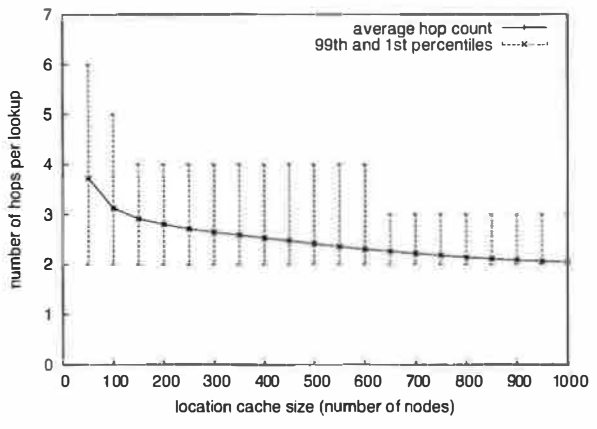
Figure 6(a) presents the results of this experiment for various location cache sizes and a 1,000 node Chord ring. Location caching reduces the number of hops to two or three because "being close counts": if the originating node, O, has not cached the target of a lookup, T, O is nonetheless likely to know about a node P near T, and P is likely to know about T. We believe that caching constant fractions of the Chord ring is reasonable because the number of nodes in a deployed SFR infrastructure would be bounded.

We must be careful, however. If the DHT's membership often changes, larger location caches could have more stale state and thus be detrimental. We expect, though, that as long as the membership changes relatively infrequently compared to the rate of requests, then failures will not much alter hop counts or even latencies. To see why, note that if a DHT node O attempts to communicate with a failed node, F, O will wait for a set length of time (500 ms in the simulation) before concluding that F is inaccessible. After this interval, O evicts F's entry from its cache and then tries to contact a different node. This permits lookups to make progress, even if the lookup path reaches a failed node [31]. If requests arrive frequently, then stale state will be corrected frequently and the number of timeouts will be relatively small; on *average*, therefore, the churn does not increase latency much. (This "cleaning-on-demand" is in addition to Chord's stabilization procedure, which also helps clear old state in the location cache.)

To examine failures experimentally, we first used the NLANR trace to warm up the cache with a million requests. (Although this warm-up period differs from the previous one, the effect is negligible in practice.) We then used interleaved Poisson processes: node deaths and births each occur on average once every 10 seconds, and, concurrently, 10,000 document lookups occur at an average rate of 20 per second, the approximate lookup rate in



(a)



(b)

Figure 6: Simulated effect of location caching on hop count (a) without failures and (b) with node failures. 1000 nodes.

the NLANR trace. We believe that a failure in the infrastructure every 10 seconds is a significant over-estimate. Figure 6(b) shows the results: failures do not affect average hop counts and only slightly affect the 99th and 1st percentiles. Timeouts vary slightly with the location cache size but are never more frequent than an average of .04 timeouts per lookup, with a 99th percentile of two.

We conclude this section by noting that DNS's failure resilience depends on proper manual configuration of name servers and on zone transfers between primary and secondary servers. We do not envision the same degree of manual involvement in SFR's operation.

7 Alternatives and Related Work

In this section, we consider alternative designs and related work, first focusing on design decisions we made in the context of SFR and then discussing other proposals with similar goals.

7.1 SFR with More or Less

SFR−−. We considered a more minimal design, called SFR−−, that associates every `SFRTag` to an authoritative domain that hosts the actual `o-record`. In this model, looking up an `SFRTag` in the global SFR infrastructure returns only a pointer to an organization's resolver, and then this organization-specific resolver maps the `SFRTag` to the object's actual meta-data. This approach unfortunately requires each organization to host its own SFR service and each client to do an extra lookup (if caching fails), namely the one inside the organization.

However, there are several advantages to this approach, and they are instructive. First, SFR−− is analogous to the way DNS's top-level domain servers point to NS records, so SFR−− inherits the usual benefits of hierarchy (*e.g.*, fate sharing), but it does so without any structure built into the references themselves. Second, SFR−−'s global records point only to individual organizations and so would rarely change, and third, because SFR−− offloads many of the reference resolution problems to the individual organizations, it explicitly allows each organization to implement its own solutions to problems like object migration and replication.

SFR++. SFR can't directly handle massive replication because sending all locations to the client is unwieldy, and SFR itself doesn't have any application-independent way of selecting which locations are best for the client. A modified design, SFR++, would allow SFR to disambiguate between multiple locations based on *selector fields*. That is, content providers could associate several logical `o-records` to the same `SFRTag`; when a client does a lookup on a given `SFRTag`, the infrastructure could use a client attribute, such as IP address, as a selection mechanism for choosing from a set of `o-records` the one that corresponds to a location near the client.

The ability to disambiguate based on selector fields would also allow SFR to deal more gracefully with object *transformation*, when the object referred to by a given `SFRTag` splits into several component objects. Currently SFR uses redirection (see Section 4.1), which is not ideal; with SFR-embedded disambiguation, however, clients could submit, on a lookup, the `optional_path` component of an SFR URL in addition to the `SFRTag`. The responsible SFR node could then do longest prefix matches to track transformed objects, according to a table set by the content provider and stored alongside the `o-record`.

7.2 RRS: Other Approaches

The Globe literature [2, 32, 33] articulates the case for a single, general-purpose infrastructure for mapping persistent object identifiers to current locations. While they do not state that references should be inherently human-unfriendly, they do observe (1) that persistence implies that references cannot encode information about how they are resolved and (2) that human-level names should be strictly separated from identifiers. Globe's choice of a resolving substrate, however, differs from ours; in particular, the Globe location service relies on distributed trees overlaid on a static hierarchy of nodes (though, as in SFR, the identifiers themselves are not hierarchical nor is there any *a priori* difference among the hosts comprising the tree). SFR's approach to reference integrity, fate sharing, and latency differ from Globe's as well.

The URN community [2, 5, 9, 19, 28, 29] makes a case nearly identical to Globe's for persistent identifiers that identify individual objects; they propose a framework in which each application would have its own resolving infrastructure and its own namespace. In addition, the URN standards specify that references are to be human-unfriendly [29], but they neither specifically advocate that the infrastructure enforce randomness in the references nor do they propose a way to resolve these references.

O'Donnell articulates the need for a human-unfriendly namespace with persistent identifiers, and his vision is similar to ours [23, 24]. However, his numeric Open Network Handles would each exist in their own DNS domain underneath particular altruistic providers (*e.g.*, `h1282132.nicesponsor.org`); this approach contrasts with our claim that full location independence means not encoding any identifying information—not even about the provider responsible for the meta-data—into the reference itself.

If Open Network Handles were enhanced so the altruistic provider were removed from the URL, then all of these handles would exist in the same domain, and the challenge of routing in an unpartitioned namespace would arise, along with the other challenges we mention. This scheme would thus be functionally equivalent to SFR. (This assumes DNS were augmented with a record type that abstracted individual objects, otherwise these Handles could not provide location-independent, per-object references.)

As a final alternative, the Secure File System (SFS) [18] is an example of an existing system that relies on human-unfriendly identifiers with cryptographic guarantees. Although SFS references consist of a hostname, a hash of a public key, and a pathname, and are thereby tied to administrative domains, one could extend SFS to provide machine independent references by, for example, removing the hostname component and mapping the hash of the public key to a machine via a level of indirection like our `o-record`. At that point, SFS would either face similar challenges to the ones we have identified, or it would make use of SFR (though it would additionally have all of SFS's security benefits). However, this scheme could not provide any kind of object migration without redirects (implemented via symbolic links in SFS space), and the SFS literature has never articulated the need for persistent, location-independent identifiers.

7.3 Other Related Work

Frankston notes DNS's conflation of user-level names and references and also proposes a set of semantic-free references for the Web, though he does not detail a design [10]. The PURL project provides a layer of indirection via HTTP redirects to give location-independent, persistent URLs that may or may not contain semantics [27]. Phelps and Wilensky suggest a scheme for robust hyperlinks in which every document would have a unique signature, consisting of several words, and every referring hyperlink would embed the signature so that if a link were broken, a search engine could then find the document [25]. These schemes make use of, and are therefore constrained by, the existing Web infrastructure.

Digital Object Identifiers (DOIs) [13] are a URN implementation with persistent object identifiers in a managed but human-unfriendly namespace. DOIs are in use (including by ACM) and rely on the Handle System [6], an RRS that maps persistent identifiers to object meta-data using two levels of hierarchy.

The *i3* infrastructure envisions a widely deployed substrate for a general form of indirection [30]. This service indirects *routing* whereas SFR is an application-level layer of indirection for *naming*. Cox *et al.* describe an implementation of DNS in which they use Chord as a lookup mechanism for DNS A-records, thereby eliminating many administrative problems that result from the hierarchy in DNS [7]. They hash domain names into a flat namespace and use the original names both as identifiers and as a way of creating a public key hierarchy to authenticate a given A-record. They do not assume widespread caching of either the data being delivered (`o-records` in our case, RRsets in theirs) or of the other nodes in the Chord ring. Based on pessimistic assumptions about the infrastructure (ones we do not share because we think our system will be a managed service in which locations are cached), they conclude that the performance of DNS over Chord is unacceptable. They make no arguments in favor of an application-independent, semantic-free, general purpose referencing infrastructure and envision using current DNS names as references.

8 Conclusion

The goal of SFR is not to provide equivalent functionality to DNS, which ought to continue with its original purpose of hostname translation, but rather to provide a more attractive alternative for the subclass of applications, like referencing Web objects, that require an RRS.

In this paper, we have knowingly adopted an extreme view, namely that references should encode neither human readable semantics nor any other information about the referenced object. It is entirely possible, however, that the referencing system of the future will be somewhere in between DNS and SFR, either because human readability turns out to be critical or because a hierarchical resolving scheme that ties references to particular providers turns out to be the right economic model. For now, we simply observe that from a usability perspective, today's DNS and SFR each offer something the other does not. DNS makes composing and publicizing content easy while SFR attempts to achieve the full potential of the Web as a medium in which anyone can publish (even without controlling a domain), in which objects can freely migrate, and for which the infrastructure is simple, robust, and accessible.

Acknowledgments

We gratefully acknowledge: the anonymous reviewers, Brad Karp, David Andersen, Frans Kaashoek, Jaeyeon Jung, Nick Feamster, Maxwell Krohn, Bret Hull, Michael Afergan, Bob Briscoe, Karthik Lakshminarayanan, Sriram Ramabhadran, and our shepherd, Bill Weihl, who all read drafts and made many helpful suggestions; David Mazières, Bryan Ford, Karen Sollins, Jeffrey Considine, and Eddie Kohler, for useful conversations and ideas; Emil Sit, Benjie Chen, and Frank Dabek, for help with Chord; and Jaeyeon Jung for help with the DNS data. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

References

[1] H. Balakrishnan, S. Shenker, and M. Walfish. Semantic-free referencing in linked distributed systems. In *2nd International Workshop on Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.

[2] G. Ballintijn, M. van Steen, and A. S. Tanenbaum. Scalable user-friendly resource names. *IEEE Internet Computing*, 5(5):20–27, 2001.

[3] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's Internet. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.

[4] Computerworld. `http://www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,82888,00.html`.

[5] D. Connolly. Naming and addressing: URIs, URLs, ... W3C Architecture Document.

[6] Corporation for National Research Initiatives. `http://www.handle.net/`.

[7] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, Mar. 2002.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[9] L. Daigle, D. van Gulik, R. Iannella, and P. Faltstrom. URN namespace definition mechanisms, June 1999. RFC 2611.

[10] B. Frankston. DNS: A safe haven. `http://www.frankston.com/public/ESSAYS/DNSSafeHaven.asp`.

[11] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, 2004.

[12] N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, March 2003.

[13] International DOI Foundation. `http://www.doi.org/`.

[14] IRCache. `http://www.ircache.net/`. NSF (grants NCR-9616602 and NCR-9521745), National Laboratory for Applied Network Research.

[15] Infrastructure for resilient Internet systems. `http://www.project-iris.net/`, 2002.

[16] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS peformance and the effectiveness of caching. *IEEE/ACM Trans. on Networking*, 10(5), Oct. 2002.

[17] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, pages 261–274, Boston, MA, June 2001.

[18] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, Dec. 1999.

[19] R. Moats. URN syntax, May 1997. RFC 2141.

[20] P. Mockapetris. Domain Names – Concepts and Facilities, Nov 1987. RFC 1034.

[21] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1998.

[22] M. Mueller. *Ruling the Root: Internet Governance and the Taming of Cyberspace*. MIT Press, Cambridge, MA, May 2002.

[23] M. O'Donnell. Open network handles implemented in DNS, Sep. 2002. Internet Draft, draft-odonnell-onhs-imp-dns-00.txt.

[24] M. O'Donnell. A proposal to separate Internet handles from names. `http://people.cs.uchicago.edu/~odonnell/Citizen/Network_Identifiers/`, Feb 2003. submitted for publication.

[25] T. A. Phelps and R. Wilensky. Robust hyperlinks: Cheap, everywhere, now. In *Proceedings of Digital Documents and Electronic Publishing (DDEP00)*, Munich, Germany, Sept 2000.

[26] PlanetLab. `http://www.planet-lab.org`.

[27] K. Shafer, S. Weibel, E. Jul, and J. Fausey. Introduction to persistent uniform resource locators. `http://purl.oclc.org/docs/inet96.html`. OCLC Online Computer Library Center, Inc., 6565 Frantz Road, Dublin, Ohio 43017-3395.

[28] K. Sollins. Architectural principles of uniform resource name resolution, Jan 1998. RFC 2276.

[29] K. Sollins and L. Masinter. Functional requirements for uniform resource names, Dec 1994. RFC 1737.

[30] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[32] M. van Steen and G. Ballintijn. Achieving scalability in hierarchical location services. In *Proc. 26th International Computer Software and Applications Conference*, Oxford, UK, Aug. 2002.

[33] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine*, 36(1):104–109, Jan. 1998.

Democratizing content publication with Coral

Michael J. Freedman, Eric Freudenthal, David Mazières
New York University
`http://www.scs.cs.nyu.edu/coral/`

Abstract

CoralCDN is a peer-to-peer content distribution network that allows a user to run a web site that offers high performance and meets huge demand, all for the price of a cheap broadband Internet connection. Volunteer sites that run CoralCDN automatically replicate content as a side effect of users accessing it. Publishing through CoralCDN is as simple as making a small change to the hostname in an object's URL; a peer-to-peer DNS layer transparently redirects browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots that might dissuade volunteers and hurt performance. It achieves this through Coral, a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a *distributed sloppy hash table*, or DSHT.

1 Introduction

The availability of content on the Internet is to a large degree a function of the cost shouldered by the publisher. A well-funded web site can reach huge numbers of people through some combination of load-balanced servers, fast network connections, and commercial content distribution networks (CDNs). Publishers who cannot afford such amenities are limited in the size of audience and type of content they can serve. Moreover, their sites risk sudden overload following publicity, a phenomenon nicknamed the "Slashdot" effect, after a popular web site that periodically links to under-provisioned servers, driving unsustainable levels of traffic to them. Thus, even struggling content providers are often forced to expend significant resources on content distribution.

Fortunately, at least with static content, there is an easy way for popular data to reach many more people than publishers can afford to serve themselves—volunteers can mirror the data on their own servers and networks. Indeed, the Internet has a long history of organizations with good network connectivity mirroring data they consider to be of value. More recently, peer-to-peer file sharing has demonstrated the willingness of even individual broadband users to dedicate upstream bandwidth to redistribute content the users themselves enjoy. Additionally, organizations that mirror popular content reduce their downstream bandwidth utilization and improve the latency for local users accessing the mirror.

This paper describes CoralCDN, a decentralized, self-organizing, peer-to-peer web-content distribution network. CoralCDN leverages the aggregate bandwidth of volunteers running the software to absorb and dissipate most of the traffic for web sites using the system. In so doing, CoralCDN replicates content in proportion to the content's popularity, regardless of the publisher's resources—in effect democratizing content publication.

To use CoralCDN, a content publisher—or someone posting a link to a high-traffic portal—simply appends "`.nyud.net:8090`" to the hostname in a URL. Through DNS redirection, oblivious clients with unmodified web browsers are transparently redirected to nearby Coral web caches. These caches cooperate to transfer data from nearby peers whenever possible, minimizing both the load on the origin web server and the end-to-end latency experienced by browsers.

CoralCDN is built on top of a novel key/value indexing infrastructure called Coral. Two properties make Coral ideal for CDNs. First, Coral allows nodes to locate nearby cached copies of web objects without querying more distant nodes. Second, Coral prevents hot spots in the infrastructure, even under degenerate loads. For instance, if every node repeatedly stores the same key, the rate of requests to the most heavily-loaded machine is still only logarithmic in the total number of nodes.

Coral exploits overlay routing techniques recently popularized by a number of peer-to-peer distributed hash tables (DHTs). However, Coral differs from DHTs in several ways. First, Coral's locality and hot-spot prevention properties are not possible for DHTs. Second, Coral's architecture is based on clusters of well-connected machines. Clusters are exposed in the interface to higher-level software, and in fact form a crucial part of the DNS redirection mechanism. Finally, to achieve its goals, Coral provides weaker consistency than traditional DHTs. For that reason, we call its indexing abstraction a *distributed sloppy hash table*, or DSHT.

CoralCDN makes a number of contributions. It enables people to publish content that they previously could not or would not because of distribution costs. It is the first completely decentralized and self-organizing web-content distribution network. Coral, the indexing infrastructure, provides a new abstraction potentially of use to any application that needs to locate nearby instances of resources on the network. Coral also introduces an epidemic clustering algorithm that exploits distributed network measurements. Furthermore, Coral is the first peer-to-peer key/value index that can scale to many stores of the same key without hot-spot congestion, thanks to a new rate-limiting technique. Finally, CoralCDN contains the first peer-to-peer DNS redirection infrastructure, allowing the system to inter-operate with unmodified web browsers.

Measurements of CoralCDN demonstrate that it allows under-provisioned web sites to achieve dramatically higher capacity, and its clustering provides quantitatively better performance than locality-unaware systems.

The remainder of this paper is structured as follows. Section 2 provides a high-level description of CoralCDN, and Section 3 describes its DNS system and web caching components. In Section 4, we describe the Coral indexing infrastructure, its underlying DSHT layers, and the clustering algorithms. Section 5 includes an implementation overview and Section 6 presents experimental results. Section 7 describes related work, Section 8 discusses future work, and Section 9 concludes.

2 The Coral Content Distribution Network

The Coral Content Distribution Network (CoralCDN) is composed of three main parts: (1) a network of cooperative HTTP proxies that handle users' requests,[1] (2) a network of DNS nameservers for `nyucd.net` that map clients to nearby Coral HTTP proxies, and (3) the underlying Coral indexing infrastructure and clustering machinery on which the first two applications are built.

2.1 Usage Models

To enable immediate and incremental deployment, CoralCDN is transparent to clients and requires no software or plug-in installation. CoralCDN can be used in a variety of ways, including:

- **Publishers.** A web site publisher for `x.com` can change selected URLs in their web pages to "Coralized" URLs, such as `http://www.x.com.nyud.net:8090/y.jpg`.
- **Third-parties.** An interested third-party—*e.g.*, a poster to a web portal or a Usenet group—can Coralize a URL before publishing it, causing all embedded relative links to use CoralCDN as well.
- **Users.** Coral-aware users can manually construct Coralized URLs when surfing slow or overloaded web sites. All relative links and HTTP redirects are automatically Coralized.

[1] While Coral's HTTP proxy definitely provides proxy functionality, it is not an HTTP proxy in the strict RFC2616 sense; it serves requests that are syntactically formatted for an ordinary HTTP server.

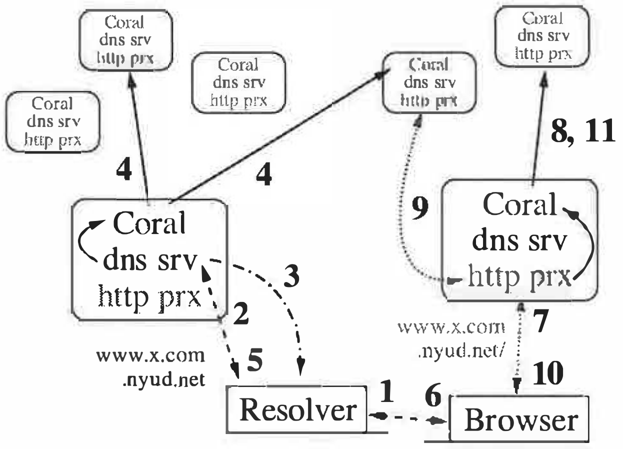


Figure 1: Using CoralCDN, the steps involved in resolving a Coralized URL and returning the corresponding file, per Section 2.2. Rounded boxes represent CoralCDN nodes running Coral, DNS, and HTTP servers. Solid arrows correspond to Coral RPCs, dashed arrows to DNS traffic, dotted-dashed arrows to network probes, and dotted arrows to HTTP traffic.

2.2 System Overview

Figure 1 shows the steps that occur when a client accesses a Coralized URL, such as `http://www.x.com.nyud.net:8090/`, using a standard web browser. The two main stages—DNS redirection and HTTP request handling—both use the Coral indexing infrastructure.

1. A client sends a DNS request for `www.x.com.nyud.net` to its local resolver.
2. The client's resolver attempts to resolve the hostname using some Coral DNS server(s), possibly starting at one of the few registered under the `.net` domain.
3. Upon receiving a query, a Coral DNS server probes the client to determines its round-trip-time and last few network hops.
4. Based on the probe results, the DNS server checks Coral to see if there are any known nameservers and/or HTTP proxies near the client's resolver.
5. The DNS server replies, returning any servers found through Coral in the previous step; if none were found, it returns a random set of nameservers and proxies. In either case, if the DNS server is close to the client, it only returns nodes that are close to itself (see Section 3.1).
6. The client's resolver returns the address of a Coral HTTP proxy for `www.x.com.nyud.net`.

7. The client sends the HTTP request `http://www.x.com.nyud.net:8090/` to the specified proxy. If the proxy is caching the file locally, it returns the file and stops. Otherwise, this process continues.
8. The proxy looks up the web object's URL in Coral.
9. If Coral returns the address of a node caching the object, the proxy fetches the object from this node. Otherwise, the proxy downloads the object from the origin server, `www.x.com` (not shown).
10. The proxy stores the web object and returns it to the client browser.
11. The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL.

2.3 The Coral Indexing Abstraction

This section introduces the Coral indexing infrastructure as used by CoralCDN. Coral provides a *distributed sloppy hash table* (DSHT) abstraction. DSHTs are designed for applications storing soft-state key/value pairs, where multiple values may be stored under the same key. CoralCDN uses this mechanism to map a variety of types of key onto addresses of CoralCDN nodes. In particular, it uses DSHTs to find Coral nameservers topologically close clients' networks, to find HTTP proxies caching particular web objects, and to locate nearby Coral nodes for the purposes of minimizing internal request latency.

Instead of one global overlay as in [5, 14, 27], each Coral node belongs to several distinct DSHTs called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT) we call the *diameter*. The system is parameterized by a fixed hierarchy of diameters known as *levels*. Every node is a member of one DSHT at each level. A group of nodes can form a level-i cluster if a high-enough fraction their pair-wise RTTs are below the level-i diameter threshold. Although Coral's implementation allows for an arbitrarily-deep DSHT hierarchy, this paper describes a three-level hierarchy with thresholds of ∞, 60 msec, and 20 msed for level-0, -1, and -2 clusters respectively. Coral queries nodes in higher-level, fast clusters before those in lower-level, slower clusters. This both reduces the latency of lookups and increases the chances of returning values stored by nearby nodes.

Coral provides the following interface to higher-level applications:

- $put(key, val, ttl, [levels])$: Inserts a mapping from the key to some arbitrary value, specifying the time-to-live of the reference. The caller may optionally specify a subset of the cluster hierarchy to restrict the operation to certain levels.
- $get(key, [levels])$: Retrieves some subset of the values stored under a key. Again, one can optionally specify a subset of the cluster hierarchy.
- $nodes(level, count, [target], [services])$: Returns *count* neighbors belonging to the node's cluster as specified by *level*. *target*, if supplied, specifies the IP address of a machine to which the returned nodes would ideally be near. Coral can probe *target* and exploit network topology hints stored in the DSHT to satisfy the request. If *services* is specified, Coral will only return nodes running the particular service, *e.g.*, an HTTP proxy or DNS server.
- $levels()$: Returns the number of levels in Coral's hierarchy and their corresponding RTT thresholds.

The next section describes the design of CoralCDN's DNS redirector and HTTP proxy—especially with regard to their use of Coral's DSHT abstraction and clustering hierarchy—before returning to Coral in Section 4.

3 Application-Layer Components

The Coral DNS server directs browsers fetching Coralized URLs to Coral HTTP proxies, attempting to find ones near the requesting client. These HTTP proxies exploit each others' caches in such a way as to minimize both transfer latency and the load on origin web servers.

3.1 The Coral DNS server

The Coral DNS server, *dnssrv*, returns IP addresses of Coral HTTP proxies when browsers look up the hostnames in Coralized URLs. To improve locality, it attempts to return proxies near requesting clients. In particular, whenever a DNS resolver (client) contacts a nearby *dnssrv* instance, *dnssrv* both returns proxies within an appropriate cluster, and ensures that future DNS requests from that client will not need to leave the cluster. Using the *nodes* function, *dnssrv* also exploits Coral's on-the-fly network measurement capabilities and stored topology hints to increase the chances of clients discovering nearby DNS servers.

More specifically, every instance of *dnssrv* is an authoritative nameserver for the domain `nyucd.net`. Assuming a 3-level hierarchy, as Coral is generally configured, *dnssrv* maps any domain name ending `http.L2.L1.L0.nyucd.net` to one or more Coral HTTP proxies. (For an $(n+1)$-level hierarchy, the domain name is extended out to `L`n in the obvious way.) Because such names are somewhat unwieldy, we established a DNS DNAME alias [4], `nyud.net`, with target `http.L2.L1.L0.nyucd.net`. Any domain name ending `nyud.net` is therefore equivalent to the same name with suffix `http.L2.L1.L0.nyucd.net`, allowing Coralized URLs to have the more concise form `http://www.x.com.nyud.net:8090/`.

dnssrv assumes that web browsers are generally close to their resolvers on the network, so that the source address of a DNS query reflects the browser's network location. This assumption holds to varying degrees, but is good enough that Akamai [12], Digital Island [6], and Mirror Image [21] have all successfully deployed commercial CDNs based on DNS redirection. The locality problem therefore is reduced to returning proxies that are near the source of a DNS request. In order to achieve locality, *dnssrv* measures its round-trip-time to the resolver and categorizes it by level. For a 3-level hierarchy, the resolver will correspond to a level 2, level 1, or level 0 client, depending on how its RTT compares to Coral's cluster-level thresholds.

When asked for the address of a hostname ending `http.L2.L1.L0.nyucd.net`, *dnssrv*'s reply contains two sections of interest: A set of addresses for the name—*answers* to the query—and a set of nameservers for that name's domain—known as the *authority* section of a DNS reply. *dnssrv* returns addresses of *CoralProxies* in the cluster whose level corresponds to the client's level categorization. In other words, if the RTT between the DNS client and *dnssrv* is below the level-i threshold (for the best i), *dnssrv* will only return addresses of Coral nodes in its level-i cluster. *dnssrv* obtains a list of such nodes with the *nodes* function. Note that *dnssrv* always returns *CoralProxy* addresses with short time-to-live fields (30 seconds for levels 0 and 1, 60 for level 2).

To achieve better locality, *dnssrv* also specifies the client's IP address as a *target* argument to *nodes*. This causes Coral to probe the addresses of the last five network hops to the client and use the results to look for clustering hints in the DSHTs. To avoid significantly delaying clients, Coral maps these network hops using a fast, built-in traceroute-like mechanism that combines concurrent probes and aggressive time-outs to minimize latency. The entire mapping process generally requires around 2 RTTs and 350 bytes of bandwidth. A Coral node caches results to avoid repeatedly probing the same client.

The closer *dnssrv* is to a client, the better its selection of *CoralProxy* addresses will likely be for the client. *dnssrv* therefore exploits the authority section of DNS replies to lock a DNS client into a good cluster whenever it happens upon a nearby *dnssrv*. As with the answer section, *dnssrv* selects the nameservers it returns from the appropriate cluster level and uses the *target* argument to exploit measurement and network hints. Unlike addresses in the answer section, however, it gives nameservers in the authority section a long TTL (one hour). A nearby *dnssrv* must therefore override any inferior nameservers a DNS client may be caching from previous queries. *dnssrv* does so by manipulating the domain for which returned nameservers are servers. To clients more distant than the level-1 timing threshold, *dnssrv* claims to return nameservers for domain `L0.nyucd.net`. For clients closer than that threshold, it returns nameservers for `L1.L0.nyucd.net`. For clients closer than the level-2 threshold, it returns nameservers for domain `L2.L1.L0.nyucd.net`. Because DNS resolvers query the servers for the most specific known domain, this scheme allows closer *dnssrv* instances to override the results of more distant ones.

Unfortunately, although resolvers can tolerate a fraction of unavailable DNS servers, browsers do not handle bad HTTP servers gracefully. (This is one reason for returning *CoralProxy* addresses with short TTL fields.) As an added precaution, *dnssrv* only returns *CoralProxy* addresses which it has recently verified first-hand. This sometimes means synchronously checking a proxy's status (via a UDP RPC) prior replying to a DNS query. We note further that people who wish to contribute only upstream bandwidth can flag their proxy as "non-recursive," in which case *dnssrv* will only return that proxy to clients on local networks.

3.2 The Coral HTTP proxy

The Coral HTTP proxy, *CoralProxy*, satisfies HTTP requests for Coralized URLs. It seeks to provide reasonable request latency and high system throughput, even while serving data from origin servers behind comparatively slow network links such as home broadband connections. This design space requires particular care in minimizing load on origin servers compared to traditional CDNs, for two reasons. First, many of Coral's origin servers are likely to have slower network connections than typical customers of commercial CDNs. Second, commercial CDNs often collocate a number of machines at each deployment site and then select proxies based in part on the URL requested—effectively distributing URLs across proxies. Coral, in contrast, selects proxies only based on client locality. Thus, in CoralCDN, it is much easier for every single proxy to end up fetching a particular URL.

To aggressively minimize load on origin servers, a *CoralProxy* must fetch web pages from other proxies whenever possible. Each proxy keeps a local cache from which it can immediately fulfill requests. When a client requests a non-resident URL, *CoralProxy* first attempts to locate a cached copy of the referenced resource using Coral (a *get*), with the resource indexed by a SHA-1 hash of its URL [22]. If *CoralProxy* discovers that one or more other proxies have the data, it attempts to fetch the data from the proxy to which it first connects. If Coral provides no referrals or if no referrals return the data, *CoralProxy* must fetch the resource directly from the origin.

While *CoralProxy* is fetching a web object—either from the origin or from another *CoralProxy*—it inserts a reference to itself in its DSHTs with a time-to-live of 20 seconds. (It will renew this short-lived reference until it completes the download.) Thus, if a flash crowd suddenly fetches a web page, all *CoralProxies*, other than the first simultaneous requests, will naturally form a kind of multicast tree for retrieving the web page. Once any *CoralProxy* obtains the full file, it inserts a much longer-lived reference to itself (*e.g.*, 1 hour). Because the insertion algorithm accounts for TTL, these longer-lived references will overwrite shorter-lived ones, and they can be stored on well-selected nodes even under high insertion load, as later described in Section 4.2.

CoralProxies periodically renew referrals to resources in their caches. A proxy should not evict a web object from its cache while a reference to it may persist in the DSHT. Ideally, proxies would adaptively set TTLs based on cache capacity, though this is not yet implemented.

4 Coral: A Hierarchical Indexing System

This section describes the Coral indexing infrastructure, which CoralCDN leverages to achieve scalability, self-organization, and efficient data retrieval. We describe how Coral implements the *put* and *get* operations that form the basis of its *distributed sloppy hash table* (DSHT) abstraction: the underlying key-based routing layer (4.1), the DSHT algorithms that balance load (4.2), and the changes that enable latency and data-placement optimizations within a hierarchical set of DSHTs (4.3). Finally, we describe the clustering mechanisms that manage this hierarchical structure (4.4).

4.1 Coral's Key-Based Routing Layer

Coral's keys are opaque 160-bit ID values; nodes are assigned IDs in the same 160-bit identifier space. A node's ID is the SHA-1 hash of its IP address. Coral defines a distance metric on IDs. Henceforth, we describe a node as being *close* to a key if the distance between the key and the node's ID is small. A Coral *put* operation stores a key/value pair at a node close to the key. A *get* operation searches for stored key/value pairs at nodes successively closer to the key. To support these operations, a node requires some mechanism to discover other nodes close to any arbitrary key.

Every DSHT contains a routing table. For any key k, a node R's routing table allows it to find a node closer to k, unless R is already the closest node. These routing tables are based on Kademlia [17], which defines the distance between two values in the ID-space to be their bitwise exclusive or (XOR), interpreted as an unsigned integer. Using the XOR metric, IDs with longer matching prefixes (of most significant bits) are numerically *closer*.

The size of a node's routing table in a DSHT is logarithmic in the total number of nodes comprising the DSHT. If a node R is not the closest node to some key k, then R's routing table almost always contains either the closest node to k, or some node whose distance to k is at least one bit shorter than R's. This permits R to visit a sequence of nodes with monotonically decreasing distances $[d_1, d_2, \ldots]$ to k, such that the encoding of d_{i+1} as a binary number has one fewer bit than d_i. As a result, the expected number of iterations for R to discover the closest node to k is logarithmic in the number of nodes.

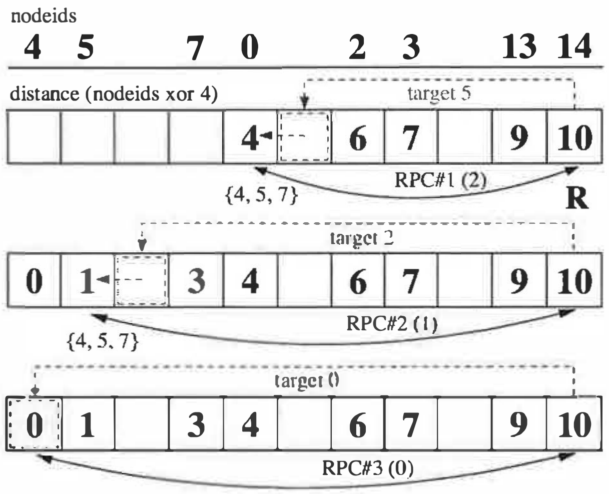


Figure 2: Example of routing operations in a system containing eight nodes with IDs $\{4, 5, 7, 0, 2, 3, 13, 14\}$. In this illustration, node R with $id = 14$ is looking up the node closest to key $k = 4$, and we have sorted the nodes by their distance to k. The top boxed row illustrates XOR distances for the nodes $\{0, 2, 3, 13, 14\}$ that are initially known by R. R first contacts a known peer whose distance to k is closest to half of R's distance $(10/2 = 5)$; in this illustration, this peer is node zero, whose distance to k is $0 \oplus 4 = 4$. Data in RPC requests and responses are shown in parentheses and braces, respectively: R asks node zero for its peers that are half-way closer to k, *i.e.*, those at distance $\frac{4}{2} = 2$. R inserts these new references into its routing table (middle row). R now repeats this process, contacting node fi ve, whose distance 1 is closest to $\frac{4}{2}$. Finally, R contacts node four, whose distance is 0, and completes its search (bottom row).

Figure 2 illustrates the Coral routing algorithm, which successively visits nodes whose distances to the key are approximately halved each iteration. Traditional key-based routing layers attempt to route directly to the node closest to the key whenever possible [25, 26, 31, 35], resorting to several intermediate hops only when faced with incomplete routing information. By caching additional routing state—beyond the necessary $\log(n)$ references—these systems in practice manage to achieve routing in a *constant* number of hops. We observe that frequent references to the same key can generate high levels of traffic in nodes close to the key. This congestion, called *tree saturation*, was first identified in shared-memory interconnection networks [24].

To minimize tree saturation, each iteration of a Coral search prefers to correct only b bits at a time.[2] More specifically, let $\text{splice}(k, r, i)$ designate the most significant bi bits of k followed by the least significant $160 - bi$ bits of r. If node R with ID r wishes to search for key k, R first initializes a variable $t \leftarrow r$. At each iteration, R updates $t \leftarrow \text{splice}(k, t, i)$, using the smallest value of i that yields a new value of t. The next hop in the lookup path is the closest node to t that already exists in R's routing table. As described below, by limiting the use of potentially closer known hops in this way, Coral can avoid overloading any node, even in the presence of very heavily accessed keys.

The potential downside of longer lookup paths is higher lookup latency in the presence of slow or stale nodes. In order to mitigate these effects, Coral keeps a window of multiple outstanding RPCs during a lookup, possibly contacting the closest few nodes to intermediary target t.

4.2 Sloppy Storage

Coral uses a sloppy storage technique that caches key/value pairs at nodes whose IDs are close to the key being referenced. These cached values reduce hot-spot congestion and tree saturation throughout the indexing infrastructure: They frequently satisfy *put* and *get* requests at nodes other than those closest to the key. This characteristic differs from DHTs, whose *put* operations all proceed to nodes closest to the key.

The Insertion Algorithm. Coral performs a two-phase operation to insert a key/value pair. In the first, or "forward," phase, Coral routes to nodes that are successively closer to the key, as previously described. However, to avoid tree saturation, an insertion operation may terminate prior to locating the closest node to the key, in which case the key/value pair will be stored at a more distant node. More specifically, the forward phase terminates whenever the storing node happens upon another node that is both *full* and *loaded* for the key:

1. A node is or *full* with respect to some key k when it stores l values for k whose TTLs are all at least one-half of the new value.
2. A node is *loaded* with respect to k when it has received more than the maximum *leakage rate* β requests for k within the past minute.

In our experiments, $l = 4$ and $\beta = 12$, meaning that under high load, a node claims to be loaded for all but one store attempt every 5 seconds. This prevents excessive numbers of requests from hitting the key's closest nodes, yet still allows enough requests to propagate to keep values at these nodes fresh.

[2]Experiments in this paper use $b = 1$.

In the forward phase, Coral's routing layer makes repeated RPCs to contact nodes successively closer to the key. Each of these remote nodes returns (1) whether the key is loaded and (2) the number of values it stores under the key, along with the minimum expiry time of any such values. The client node uses this information to determine if the remote node can accept the store, potentially evicting a value with a shorter TTL. This forward phase terminates when the client node finds either the node closest to the key, or a node that is full and loaded with respect to the key. The client node places all contacted nodes that are not both full and loaded on a stack, ordered by XOR distance from the key.

During the reverse phase, the client node attempts to insert the value at the remote node referenced by the top stack element, *i.e.*, the node closest to the key. If this operation does not succeed—perhaps due to others' insertions—the client node pops the stack and tries to insert on the new stack top. This process is repeated until a store succeeds or the stack is empty.

This two-phase algorithm avoids tree saturation by storing values progressively further from the key. Still, eviction and the leakage rate β ensure that nodes close to the key retain long-lived values, so that live keys remain reachable: β nodes per minute that contact an intermediate node (including itself) will go on to contact nodes closer to the key. For a perfectly-balanced tree, the key's closest node receives only $\left(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil\right)$ store requests per minute, when fixing b bits per iteration.

Proof sketch. Each node in a system of n nodes can be uniquely identified by a string S of $\log n$ bits. Consider S to be a string of b-bit digits. A node will contact the closest node to the key before it contacts any other node if and only if its ID differs from the key in exactly one digit. There are $\lceil (\log n)/b \rceil$ digits in S. Each digit can take on $2^b - 1$ values that differ from the key. Every node that differs in one digit will throttle all but β requests per minute. Therefore, the closest node receives a maximum rate of $\left(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil\right)$ RPCs per minute.

Irregularities in the node ID distribution may increase this rate slightly, but the overall rate of traffic is still logarithmic, while in traditional DHTs it is linear. Section 6.4 provides supporting experimental evidence.

The Retrieval Algorithm. To retrieve the value associated with a key k, a node simply traverses the ID space with RPCs. When it finds a peer storing k, the remote peer returns k's corresponding list of values. The node terminates its search and *get* returns. The requesting client application handles these redundant references in some application-specific way, *e.g.*, *CoralProxy* contacts multiple sources in parallel to download cached content.

Multiple stores of the same key will be spread over multiple nodes. The pointers retrieved by the application are thus distributed among those stored, providing load balancing both *within* Coral and between servers using Coral.

4.3 Hierarchical Operations

For locality-optimized routing and data placement, Coral uses several *levels* of DSHTs called clusters. Each level-i cluster is named by a randomly-chosen 160-bit cluster identifier; the level-0 cluster ID is predefined as 0^{160}. Recall that a set of nodes should form a cluster if their average, pair-wise RTTs are below some threshold. As mentioned earlier, we describe a three-level hierarchy with thresholds of ∞, 60 msec, and 20 msec for level-0, -1, and -2 clusters respectively. In Section 6, we present experimental evidence to the client-side benefit of clustering.

Figure 3 illustrates Coral's hierarchical routing operations. Each Coral node has the same node ID in all clusters to which it belongs; we can view a node as projecting its presence to the same location in each of its clusters. This structure must be reflected in Coral's basic routing infrastructure, in particular to support switching between a node's distinct DSHTs midway through a lookup.[3]

The Hierarchical Retrieval Algorithm. A requesting node R specifies the starting and stopping levels at which Coral should search. By default, it initiates the *get* query on its highest (level-2) cluster to try to take advantage of network locality. If routing RPCs on this cluster hit some node storing the key k (RPC 1 in Fig. 3), the lookup halts and returns the corresponding stored value(s)—a *hit*—without ever searching lower-level clusters.

If a key is not found, the lookup will reach k's closest node C_2 in this cluster (RPC 2), signifying failure at this level. So, node R continues the search in its level-1 cluster. As these clusters are very often concentric, C_2 likely exists at the identical location in the identifier space in all clusters, as shown. R begins searching onward from C_2 in its level-1 cluster (RPC 3), having already traversed the ID-space up to C_2's prefix.

Even if the search eventually switches to the global cluster (RPC 4), the total number of RPCs required is about the same as a single-level lookup service, as a lookup continues from the point at which it left off in the identifier space of the previous cluster. Thus, (1) all lookups at the beginning are fast, (2) the system can tightly bound RPC timeouts, and (3) all pointers in higher-level clusters reference data *within* that local cluster.

The Hierarchical Insertion Algorithm. A node starts by performing a *put* on its level-2 cluster as in Section 4.2, so that other nearby nodes can take advantage of locality.

[3]We initially built Coral using the Chord [31] routing layer as a block-box; difficulties in maintaining distinct clusters and the complexity of the subsequent system caused us to scrap the implementation.

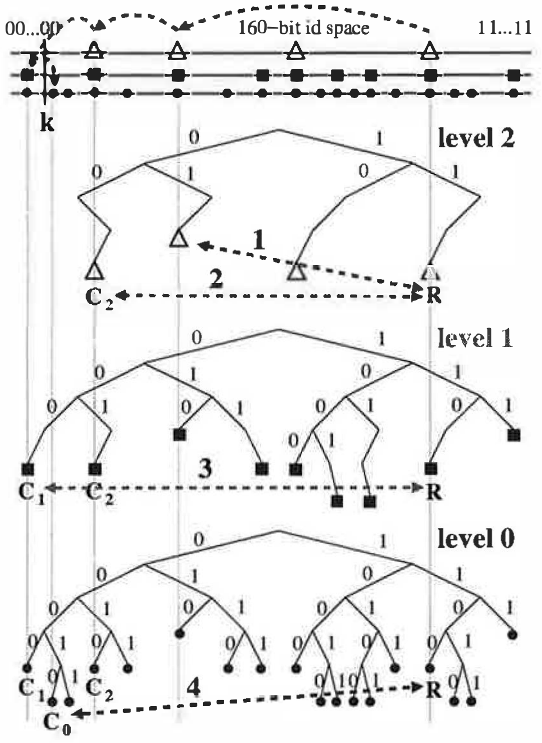


Figure 3: Coral's hierarchical routing structure. Nodes use the same IDs in each of their clusters; higher-level clusters are naturally sparser. Note that a node can be identified in a cluster by its shortest unique ID prefix, *e.g.*, "11"for R in its level-2 cluster; nodes sharing ID prefixes are located on common subtrees and are closer in the XOR metric. While higher-level neighbors usually share lower-level clusters as shown, this is not necessarily so. RPCs for a retrieval on key k are sequentially numbered.

However, this placement is only "correct" within the context of the local level-2 cluster. Thus, provided that the key is not already loaded, the node continues its insertion in the level-1 cluster from the point at which the key was inserted in level 2, much as in the retrieval case. Again, Coral traverses the ID-space only once. As illustrated in Figure 3, this practice results in a loose hierarchical cache, whereby a lower-level cluster contains nearly all data stored in the higher-level clusters to which its members also belong.

To enable such cluster-aware behavior, the headers of every Coral RPC include the sender's cluster information: the identifier, age, and a size estimate of each of its non-global clusters. The recipient uses this information to demultiplex requests properly, *i.e.*, a recipient should only consider a *put* and *get* for those levels on which it shares a cluster with the sender. Additionally, this information drives routing table management: (1) nodes are added or removed from the local cluster-specific routing tables accordingly; (2) cluster information is accumulated to drive cluster management, as described next.

4.4 Joining and Managing Clusters

As in any peer-to-peer system, a peer contacts an existing node to join the system. Next, a new node makes several queries to seed its routing tables. However, for non-global clusters, Coral adds one important requirement: A node will only join an *acceptable* cluster, where acceptability requires that the latency to 80% of the nodes be below the cluster's threshold. A node can easily determine whether this condition holds by recording minimum round-trip-times (RTTs) to some subset of nodes belonging to the cluster.

While nodes learn about clusters as a side effect of normal lookups, Coral also exploits its DSHTs to store hints. When Coral starts up, it uses its built-in fast traceroute mechanism (described in Section 3.1) to determine the addresses of routers up to five hops out. Excluding any private ("RFC1918") IP addresses, Coral uses these router addresses as keys under which to index clustering hints in its DSHTs. More specifically, a node R stores mappings from each router address to its own IP address and UDP port number. When a new node S, sharing a gateway with R, joins the network, it will find one or more of R's hints and quickly cluster with it, assuming R is, in fact, near S.

In addition, nodes store mappings to themselves using as keys any IP subnets they directly connect to and the 24-bit prefixes of gateway router addresses. These prefix hints are of use to Coral's *level* function, which traceroutes clients in the other direction; addresses on forward and reverse traceroute paths often share 24-bit prefixes.

Nodes continuously collect clustering information from peers: All RPCs include round-trip-times, cluster membership, and estimates of cluster size. Every five minutes, each node considers changing its cluster membership based on this collected data. If this collected data indicates that an alternative candidate cluster is desirable, the node first validates the collected data by contacting several nodes within the candidate cluster by routing to selected keys. A node can also form a new singleton cluster when 50% of its accesses to members of its present cluster do not meet the RTT constraints.

If probes indicate that 80% of a cluster's nodes are within acceptable TTLs and the cluster is larger, it replaces a node's current cluster. If multiple clusters are acceptable, then Coral chooses the largest cluster.

Unfortunately, Coral has only rough *approximations* of cluster size, based on its routing-table size. If nearby clusters A and B are of similar sizes, inaccurate estimations could lead to oscillation as nodes flow back-and-forth (although we have not observed such behavior). To perturb an oscillating system into a stable state, Coral employs a preference function δ that shifts every hour. A node selects the larger cluster only if the following holds:

$$\left|\log(size_A) - \log(size_B)\right| \quad > \quad \delta\,(\min(age_A, age_B))$$

where age is the current time minus the cluster's creation time. Otherwise, a node simply selects the cluster with the lower cluster ID.

We use a square wave function for δ that takes a value 0 on an even number of hours and 2 on an odd number. For clusters of disproportionate size, the selection function immediately favors the larger cluster. Otherwise, δ's transition perturbs clusters to a steady state.[4]

In either case, a node that switches clusters still remains in the routing tables of nodes in its old cluster. Thus, old neighbors will still contact it and learn of its new, potentially-better, cluster. This produces an avalanche effect as more and more nodes switch to the larger cluster. This merging of clusters is very beneficial. While a small cluster diameter provides fast lookup, a large cluster capacity increases the hit rate.

5 Implementation

The Coral indexing system is composed of a client library and stand-alone daemon. The simple client library allows applications, such as our DNS server and HTTP proxy, to connect to and interface with the Coral daemon. Coral is 14,000 lines of C++, the DNS server, *dnssrv*, is 2,000 lines of C++, and the HTTP proxy is an additional 4,000 lines. All three components use the asynchronous I/O library provided by the SFS toolkit [19] and are structured by asynchronous events and callbacks. Coral network communication is via RPC over UDP. We have successfully run Coral on Linux, OpenBSD, FreeBSD, and Mac OS X.

6 Evaluation

In this section, we provide experimental results that support our following hypotheses:

1. CoralCDN dramatically reduces load on servers, solving the "flash crowd" problem.
2. Clustering provides performance gains for popular data, resulting in good client performance.
3. Coral naturally forms suitable clusters.
4. Coral prevents hot spots within its indexing system.

[4]Should clusters of similar size continuously exchange members when δ is zero, as soon as δ transitions, nodes will all flow to the cluster with the lower cluster id. Should the clusters oscillate when $\delta = 2$ (as the estimations "hit"with one around 2^2-times larger), the nodes will all flow to the larger one when δ returns to zero.

To examine all claims, we present wide-area measurements of a synthetic work-load on CoralCDN nodes running on PlanetLab, an internationally-deployed test bed. We use such an experimental setup because traditional tests for CDNs or web servers are not interesting in evaluating CoralCDN: (1) Client-side traces generally measure the cacheability of data and client latencies. However, we are mainly interested in how well the system handles load spikes. (2) Benchmark tests such as SPECweb99 measure the web server's throughput on disk-bound access patterns, while CoralCDN is designed to reduce load on off-the-shelf web servers that are *network-bound*.

The basic structure of the experiments were is follows. First, on 166 PlanetLab machines geographically distributed mainly over North America and Europe, we launch a Coral daemon, as well as a *dnssrv* and *CoralProxy*. For experiments referred to as *multi-level*, we configure a three-level hierarchy by setting the clustering RTT threshold of level 1 to 60 msec and level 2 to 20 msec. Experiments referred to as *single-level* use only the level-0 global cluster. No objects are evicted from *CoralProxy* caches during these experiments. For simplicity, all nodes are seeded with the same well-known host. The network is allowed to stabilize for 30 minutes.[5]

Second, we run an unmodified Apache web server sitting behind a DSL line with 384 Kbit/sec upstream bandwidth, serving 12 different 41KB files, representing groups of three embedded images referenced by four web pages.

Third, we launch client processes on each machine that, after an additional random delay between 0 and 180 seconds for asynchrony, begin making HTTP GET requests to Coralized URLs. Each client generates requests for the group of three files, corresponding to a randomly selected web page, for a period of 30 minutes. While we recognize that web traffic generally has a Zipf distribution, we are attempting merely to simulate a flash crowd to a popular web page with multiple, large, embedded images (*i.e.*, the Slashdot effect). With 166 clients, we are generating 99.6 requests/sec, resulting in a cumulative download rate of approximately 32,800 Kb/sec. This rate is almost two orders of magnitude greater than the origin web server could handle. Note that this rate was chosen synthetically and in no way suggests a maximum system throughput.

For Experiment 4 (Section 6.4), we do not run any such clients. Instead, Coral nodes generate requests at very high rates, all for the same *key*, to examine how the DSHT indexing infrastructure prevents nodes close to a target ID from becoming overloaded.

[5]The stabilization time could be made shorter by reducing the clustering period (5 minutes). Additionally, in real applications, clustering is in fact a simpler task, as new nodes would immediately join nearby large clusters as they join the pre-established system. In our setup, clusters develop from an initial network comprised entirely of singletons.

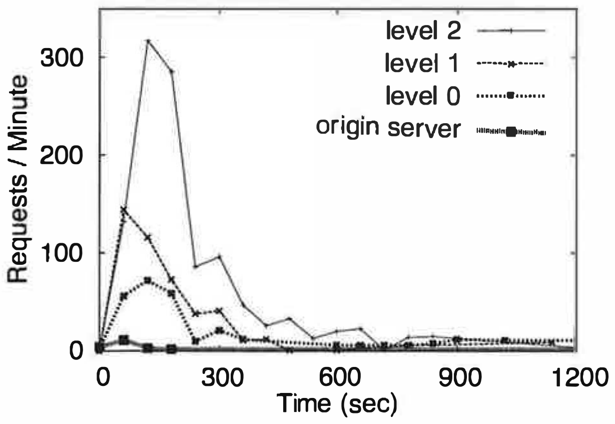


Figure 4: The number of client accesses to *CoralProxies* and the origin HTTP server. *CoralProxy* accesses are reported relative to the cluster level from which data was fetched, and do not include requests handled through local caches.

6.1 Server Load

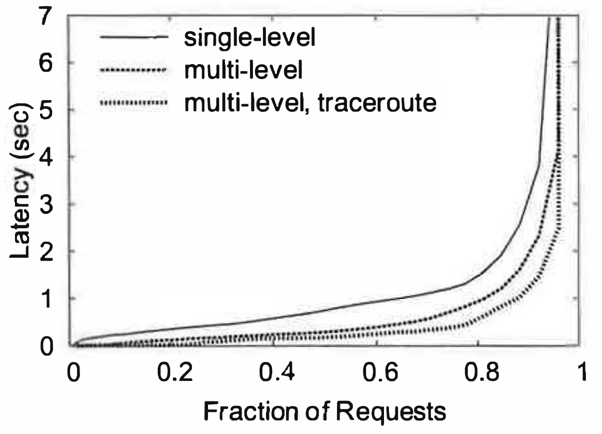
Figure 4 plots the number of requests per minute that could not be handled by a *CoralProxy*'s local cache. During the initial minute, 15 requests hit the origin web server (for 12 unique files). The 3 redundant lookups are due to the simultaneity at which requests are generated; subsequently, requests are handled either through CoralCDN's wide-area cooperative cache or through a proxy's local cache, supporting our hypothesis that CoralCDN can migrate load off of a web server.

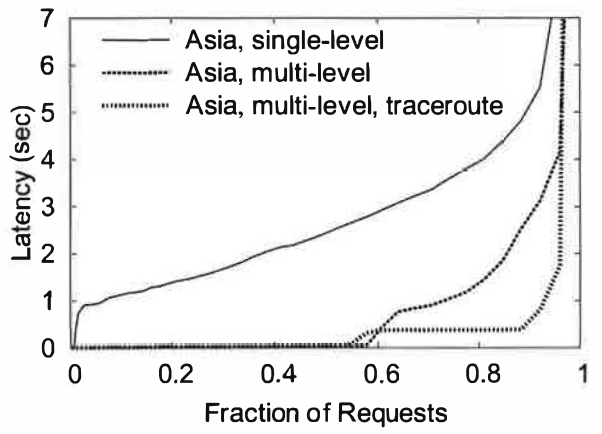
During this first minute, equal numbers of requests were handled by the level-1 and level-2 cluster caches. However, as the files propagated into *CoralProxy* caches, requests quickly were resolved within faster level-2 clusters. Within 8-10 minutes, the files became replicated at nearly every server, so few client requests went further than the proxies' local caches. Repeated runs of this experiment yielded some variance in the relative magnitudes of the initial spikes in requests to different levels, although the number of origin server hits remained consistent.

6.2 Client Latency

Figure 5 shows the end-to-end latency for a client to fetch a file from CoralCDN, following the steps given in Section 2.2. The top graph shows the latency across all PlanetLab nodes used in the experiment, the bottom graph only includes data from the clients located on 5 nodes in Asia (Hong Kong (2), Taiwan, Japan, and the Philippines). Because most nodes are located in the U.S. or Europe, the performance benefit of clustering is much more pronounced on the graph of Asian nodes.

Recall that this end-to-end latency includes the time for the client to make a DNS request and to connect to the





Request latency (sec)	All nodes		Asian nodes	
	50%	96%	50%	96%
single-level	0.79	9.54	2.52	8.01
multi-level	0.31	4.17	0.04	4.16
multi-level, traceroute	0.19	2.50	0.03	1.75

Figure 5: End-to-End client latency for requests for Coralized URLs, comparing the effect of single-level vs. multi-level clusters and of using traceroute during DNS redirection. The top graph includes all nodes; the bottom only nodes in Asia.

discovered *CoralProxy*. The proxy attempts to fulfill the client request first through its local cache, then through Coral, and finally through the origin web server. We note that *CoralProxy* implements cut-through routing by forwarding data to the client prior to receiving the entire file.

These figures report three results: (1) the distribution of latency of clients using only a single level-0 cluster (the solid line), (2) the distribution of latencies of clients using multi-level clusters (dashed), and (3) the same hierarchical network, but using traceroute during DNS resolution to map clients to nearby proxies (dotted).

All clients ran on the same subnet (and host, in fact) as a *CoralProxy* in our experimental setup. This would not be the case in the real deployment: We would expect a combination of hosts sharing networks with *CoralProxies*—within the same IP prefix as registered with Coral—and hosts without. Although the multi-level network using traceroute provides the lowest latency at most percentiles, the multi-level system without traceroute also performs better than the single-level system. Clustering has a clear performance benefit for clients, and this benefit is particularly apparent for poorly-connected hosts.

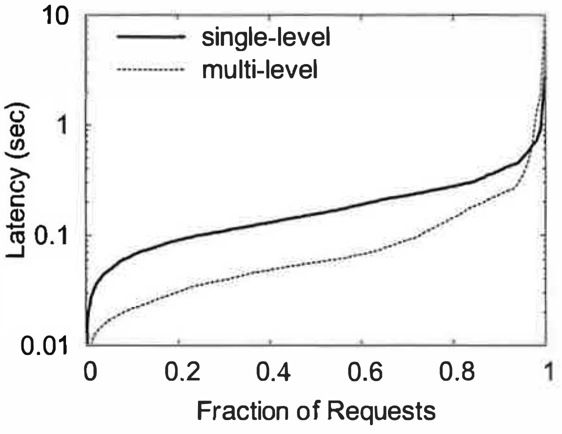
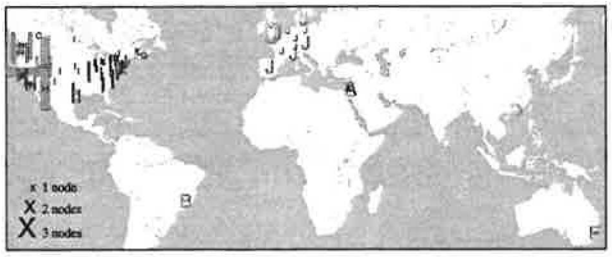


Figure 6: Latencies for proxy to *get* keys from Coral.

Figure 6 shows the latency of *get* operations, as seen by *CoralProxies* when they lookup URLs in Coral (Step 8 of Section 2.2). We plot the *get* latency on the single level-0 system vs. the multi-level systems. The multi-level system is 2-5 times faster up to the 80% percentile. After the 98% percentile, the single-level system is actually faster: Under heavy packet loss, the multi-system requires a few more timeouts as it traverses its hierarchy levels.

6.3 Clustering

Figure 7 illustrates a snapshot of the clusters from the previous experiments, at the time when clients began fetching URLs (30 minutes out). This map is meant to provide a qualitative feel for the organic nature of cluster development, as opposed to offering any quantitative measurements. On both maps, each unique, non-singleton cluster within the network is assigned a letter. We have plotted the location of our nodes by latitude/longitude coordinates. If two nodes belong to the same cluster, they are represented by the same letter. As each PlanetLab site usually collocates several servers, the size of the letter expresses the number of nodes at that site that belong to the same cluster. For example, the very large "H" (world map) and "A" (U.S. map) correspond to nodes collocated at U.C. Berkeley. We did not include singleton clusters on the maps to improve readability; post-run analysis showed that such nodes' RTTs to others (surprisingly, sometimes even at the same site) were above the Coral thresholds.



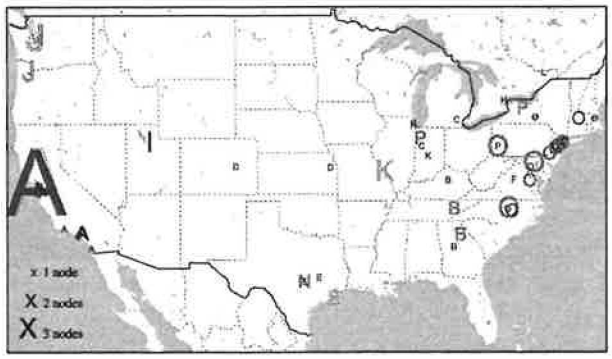


Figure 7: World view of level-1 clusters (60 msec threshold), and United States view of level-2 clusters (20 msec threshold). Each unique, non-singleton cluster is assigned a letter; the size of the letter corresponds to collocated nodes in the same cluster.

The world map shows that Coral found natural divisions between sets of nodes along geospatial lines at a 60 msec threshold. The map shows several distinct regions, the most dramatic being the Eastern U.S. (70 nodes), the Western U.S. (37 nodes), and Europe (19 nodes). The close correlation between network and physical distance suggests that speed-of-light delays dominate round-trip-times. Note that, as we did not plot singleton clusters, the map does not include three Asian nodes (in Japan, Taiwan, and the Philippines, respectively).

The United States map shows level-2 clusters again roughly separated by physical locality. The map shows 16 distinct clusters; obvious clusters include California (22 nodes), the Pacific Northwest (9 nodes), the South, the Midwest, etc. The Northeast Corridor cluster contains 29 nodes, stretching from North Carolina to Massachusetts. One interesting aspect of this map is the three separate, non-singleton clusters in the San Francisco Bay Area. Close examination of individual RTTs between these sites shows widely varying latencies; Coral clustered correctly given the underlying network topology.

6.4 Load Balancing

Finally, Figure 8 shows the extent to which a DSHT balances requests to the same key ID. In this experiment, we ran 3 nodes on each of the earlier hosts for a total of 494 nodes. We configured the system as a single level-0 cluster. At the same time, all PlanetLab nodes began to issue back-to-back *put*/*get* requests at their maximum (non-concurrent) rates. All operations referenced the same key; the values stored during *put* requests were randomized. On average, each node issued 400 *put*/*get* operation pairs per second, for a total of approximately 12 million *put*/*get* requests per minute, although only a fraction hit the network. Once a node is storing a key, *get* requests are satisfied locally. Once it is *loaded*, each node only allows the leakage rate β RPCs "through" it per minute.

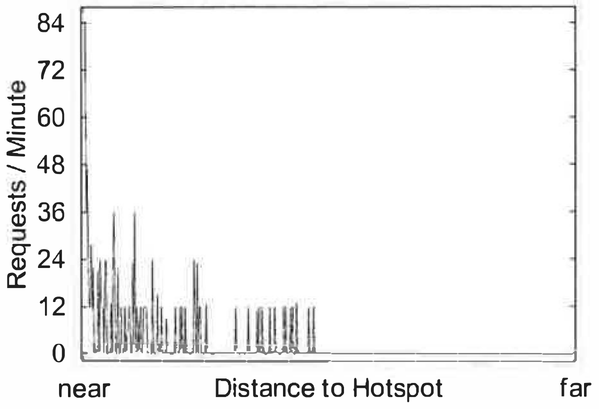


Figure 8: The total number of *put* RPCs hitting each Coral node per minute, sorted by distance from node ID to target key.

The graphs show the number of *put* RPCs that hit each node in steady-state, sorted by the XOR distance of the node's ID to the key. During the first minute, the closest node received 106 *put* RPCs. In the second minute, as shown in Figure 8, the system reached steady-state with the closest node receiving 83 *put* RPCs per minute. Recall that our equation in Section 4.2 predicts that it should receive $(\beta \cdot \log n) = 108$ RPCs per minute. The plot strongly emphasizes the efficacy of the leakage rate $\beta = 12$, as the number of RPCs received by the majority of nodes is a low multiple of 12.

No nodes on the far side of the graph received any RPCs. Coral's routing algorithm explains this condition: these nodes begin routing by flipping their ID's most-significant bit to match the *key*'s, and they subsequently contact a node on the near side. We have omitted the graph of *get* RPCs: During the first minute, the most-loaded node received 27 RPCs; subsequently, the key was widely distributed and the system quiesced.

7 Related work

CoralCDN builds on previous work in peer-to-peer systems and web-based content delivery.

7.1 DHTs and directory services

A *distributed hash table* (DHT) exposes two basic functions to the application: $put(key, value)$ stores a value at the specified key ID; $get(key)$ returns this stored value, just as in a normal hash table. Most DHTs use a key-based routing layer—such as CAN [25], Chord [31], Kademlia [17], Pastry [26], or Tapestry [35]—and store keys on the node whose ID is closest to the key. Keys must be well distributed to balance load among nodes. DHTs often replicate multiply-fetched key/value pairs for scalability, *e.g.*, by having peers replicate the pair onto the second-to-last peer they contacted as part of a *get* request.

DHTs can act either as actual data stores or merely as directory services storing pointers. CFS [5] and PAST [27] take the former approach to build a distributed file system: They require true read/write consistency among operations, where writes should atomically replace previously-stored values, not modify them.

Using the network as a directory service, Tapestry [35] and Coral relax the consistency of operations in the network. To put a key, Tapestry routes along fast hops between peers, placing at each peer a pointer back to the sending node, until it reaches the node closest to the key. Nearby nodes routing to the same key are likely to follow similar paths and discover these cached pointers. Coral's flexible clustering provides similar latency-optimized lookup and data placement, and its algorithms prevent multiple stores from forming hot spots. SkipNet also builds a hierarchy of lookup groups, although it explicitly groups nodes by domain name to support organizational disconnect [9].

7.2 Web caching and content distribution

Web caching systems fit within a large class of CDNs that handle high demand through diverse replication.

Prior to the recent interest in peer-to-peer systems, several projects proposed cooperative Web caching [2, 7, 8, 16]. These systems either multicast queries or require that caches know some or all other servers, which worsens their scalability, fault-tolerance, and susceptibility to hot spots. Although the cache hit rate of cooperative web caching increases only to a certain level, corresponding to a moderate population size [34], highly-scalable cooperative systems can still increase the total system throughput by reducing server-side load.

Several projects have considered peer-to-peer overlays for web caching, although all such systems only benefit participating clients and thus require widespread adoption to reduce server load. Stading *et al.* use a DHT to cache replicas [29], and PROOFS uses a randomized overlay to distribute popular content [30]. Both systems focus solely on mitigating flash crowds and suffer from high request latency. Squirrel proposes web caching on a traditional DHT, although only for organization-wide networks [10]. Squirrel reported poor load-balancing when the system stored pointers in the DHT. We attribute this to the DHT's inability to handle too many values for the same key—Squirrel only stored 4 pointers per object—while CoralCDN references many more proxies by storing different sets of pointers on different nodes. SCAN examined replication policies for data disseminated through a multicast tree from a DHT deployed at ISPs [3].

Akamai [1] and other commercial CDNs use DNS redirection to reroute client requests to local clusters of machines, having built detailed maps of the Internet through a combination of BGP feeds and their own measurements, such as traceroutes from numerous vantage points [28]. Then, upon reaching a cluster of collocated machines, hashing schemes [11, 32] map requests to specific machines to increase capacity. These systems require deploying large numbers of highly provisioned servers, and typically result in very good performance (both latency and throughput) for customers.

Such centrally-managed CDNs appear to offer two benefits over CoralCDN. (1) CoralCDN's network measurements, via traceroute-like probing of DNS clients, are somewhat constrained in comparison. CoralCDN nodes do not have BGP feeds and are under tight latency constraints to avoid delaying DNS replies while probing. Additionally, Coral's design assumes that no single node even knows the identity of all other nodes in the system, let alone their precise network location. Yet, if many people adopt the system, it will build up a rich database of neighboring networks. (2) CoralCDN offers less aggregate storage capacity, as cache management is completely localized. But, it is designed for a much larger number of machines and vantage points: CoralCDN may provide better performance for small organizations hosting nodes, as it is not economically efficient for commercial CDNs to deploy machines behind most bottleneck links.

More recently, CoDeeN has provided users with a set of open web proxies [23]. Users can reconfigure their browsers to use a CoDeeN proxy and subsequently enjoy better performance. The system has been deployed, and anecdotal evidence suggests it is very successful at distributing content efficiently. Earlier simulation results show that certain policies should achieve high system throughput and low request latency [33]. (Specific details of the deployed system have not yet been published, including an Akamai-like service also in development.)

Although CoDeeN gives *participating* users better performance to *most* web sites, CoralCDN's goal is to gives *most* users better performance to *participating* web sites—namely those whose publishers have "Coralized" the URLs. The two design points pose somewhat different challenges. For instance, CoralCDN takes pains to greatly minimize the load on under-provisioned origin servers, while CoDeeN has tighter latency requirements as it is on the critical path for *all* web requests. Finally, while CoDeeN has suffered a number of administrative headaches, many of these problems do not apply to CoralCDN, as, *e.g.*, CoralCDN does not allow POST operations or SSL tunneling, and it can be barred from accessing particular sites without affecting users' browsing experience.

8 Future Work

Security. This paper does not address CoralCDN's security issues. Probably the most important issue is ensuring the integrity of cached data. Given our experience with spam on the Internet, we should expect that adversaries will attempt to replace cached data with advertisements for pornography or prescription drugs. A solution is future work, but breaks down into three components.

First, honest Coral nodes should not cache invalid data. A possible solution might include embedding self-certifying pathnames [20] in Coralized URLs, although this solution requires server buy-in. Second, Coral nodes should be able to trace the path that cached data has taken and exclude data from known bad systems. Third, we should try to prevent clients from using malicious proxies. This requires client buy-in, but offers additional incentives for organizations to run Coral: Recall that a client will access a local proxy when one is available, or administrators can configure a local DNS resolver to always return a *specific* Coral instance. Alternatively, "SSL splitting" [15] provides end-to-end security between clients and servers, albeit at a higher overhead for the origin servers.

CoralCDN may require some additional abuse-prevention mechanisms, such as throttling bandwidth hogs and restricting access to address-authenticated content [23]. To leverage our redundant resources, we are considering efficient erasure coding for large-file transfers [18]. For such, we have developed on-the-fly verification mechanisms to limit malicious proxies' abilities to waste a node's downstream bandwidth [13].

Leveraging the Clustering Abstraction. This paper presents clustering mainly as a performance optimization for lookup operations and DNS redirection. However, the clustering algorithms we use are driven by *generic* policies that could allow hierarchy creation based on a variety of criteria. For example, one could provide a clustering policy by IP routing block or by AS name, for a simple mechanism that reflects administrative control and performs well under network partition. Or, Coral's clusters could be used to explicitly encode a web-of-trust security model in the system, especially useful given its standard open-admissions policy. Then, clusters could easily represent trust relationships, allowing lookups to resolve at the most trustworthy hosts. Clustering may prove to be a very useful abstraction for building interesting applications.

Multi-cast Tree Formation. CoralCDN may transmit multiple requests to an origin HTTP server at the beginning of a flash crowd. This is caused by a race condition at the key's closest node, which we could eliminate by extending store transactions to provide return status information (like test-and-set in shared-memory systems). Similar extensions to store semantics may be useful for balancing its dynamically-formed dissemination trees.

Handling Heterogeneous Proxies. We should consider the heterogeneity of proxies when performing DNS redirection and intra-Coral HTTP fetches. We might use some type of feedback-based allocation policy, as proxies can return their current load and bandwidth availability, given that they are already probed to determine liveness.

Deployment and Scalability Studies. We are planning an initial deployment of CoralCDN as a long-lived PlanetLab port 53 (DNS) service. In doing so, we hope to gather measurements from a large, active client population, to better quantify CoralCDN's scalability and effectiveness: Given our client-transparency, achieving wide-spread use is much easier than with most peer-to-peer systems.

9 Conclusions

CoralCDN is a peer-to-peer web-content distribution network that harnesses people's willingness to redistribute data they themselves find useful. It indexes cached web content with a new distributed storage abstraction called a DSHT. DSHTs map a key to multiple values and can scale to many stores of the same key without hot-spot congestion. Coral successfully clusters nodes by network diameter, ensuring that nearby replicas of data can be located and retrieved without querying more distant nodes. Finally, a peer-to-peer DNS layer redirects clients to nearby *CoralProxies*, allowing unmodified web browsers to benefit from CoralCDN, and more importantly, to avoid overloading origin servers.

Measurements of CoralCDN demonstrate that it allows under-provisioned web sites to achieve dramatically higher capacity. A web server behind a DSL line experiences hardly any load when hit by a flash crowd with a sustained aggregate transfer rate that is two orders of magnitude greater than its bandwidth. Moreover, Coral's clustering mechanism forms qualitatively sensible geographic clusters and provides quantitatively better performance than locality-unaware systems.

We have made CoralCDN freely available, so that even people with slow connections can publish web sites whose capacity grows automatically with popularity. Please visit `http://www.scs.cs.nyu.edu/coral/`.

Acknowledgments. We are grateful to Vijay Karamcheti for early conversations that helped shape this work. We thank David Andersen, Nick Feamster, Daniel Giffin, Robert Grimm, and our shepherd, Marvin Theimer, for their helpful feedback on drafts of this paper. Petar Maymounkov and Max Krohn provided access to Kademlia data structure and HTTP parsing code, respectively. We thank the PlanetLab support team for allowing us the use of UDP port 53 (DNS), despite the additional hassle this caused them. Coral is part of project IRIS (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. David Mazières is supported by an Alfred P. Sloan Research Fellowship. Michael Freedman is supported by an NDSEG Fellowship.

References

[1] Akamai Technologies, Inc. http://www.akamai.com/, 2004.

[2] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *USENIX*, Jan 1996.

[3] Y. Chen, R. Katz, and J. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proceedings of the International Conference on Pervasive Computing*, Zurich, Switzerland, Aug 2002.

[4] M. Crawford. RFC 2672: Non-terminal DNS name redirection, Aug 1999.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Banff, Canada, Oct 2001.

[6] Digital Island, Inc. http://www.digitalisland.com/, 2004.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web-cache sharing protocol. Technical Report 1361, CS Dept, U. Wisconson, Madison, Feb 1998.

[8] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Perf.*, Madison, WI, Jun 1998.

[9] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, Seattle, WA, Mar 2003.

[10] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *PODC*, Monterey, CA, Jul 2002.

[11] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[12] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *WWW8 / Computer Networks*, 31(11–16):1203–1213, 1999.

[13] M. Krohn, M. J. Freedman, and D. Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *IEEE Symp. on Security and Privacy*, Oakland, CA, May 2004.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, Cambridge, MA, Nov 2000.

[15] C. Lesniewski-Laas and M. F. Kaashoek. SSL splitting: Securely serving data from untrusted caches. In *USENIX Security*, Washington, D.C., Aug 2003.

[16] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *WWW*, Apr 1995.

[17] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, Cambridge, MA, Mar 2002.

[18] P. Maymounkov and D. Mazières. Rateless codes and big downloads. In *IPTPS*, Berkeley, CA, Feb 2003.

[19] D. Mazières. A toolkit for user-level file systems. In *USENIX*, Boston, MA, Jun 2001.

[20] D. Mazières and M. F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *ACM SIGOPS European Workshop*, Sep 1998.

[21] Mirror Image Internet. http://www.mirror-image.com/, 2004.

[22] *FIPS Publication 180-1: Secure Hash Standard*. National Institute of Standards and Technology (NIST), Apr 1995.

[23] V. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets*, Cambridge, MA, Nov 2003.

[24] G. Pfister and V. A. Norton. "hotspot" contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, 34(10), Oct 1985.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, Aug 2001.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Nov 2001.

[27] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, Banff, Canada, Oct 2001.

[28] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM*, Pittsburgh, PA, Aug 2002.

[29] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flashcrowds. In *IPTPS*, Cambridge, MA, Mar 2002.

[30] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flashcrowds. In *IEEE ICNP*, Paris, France, Nov 2002.

[31] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[32] D. Thaler and C. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. on Networking*, 6(1):1–14, 1998.

[33] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on cdn robustness. In *OSDI*, Boston, MA, Dec 2002.

[34] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, Kiawah Island, SC, Dec 1999.

[35] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 2003.

Operating System Support for Planetary-Scale Network Services

Andy Bavier* Mic Bowman† Brent Chun† David Culler‡ Scott Karlin* Steve Muir*
Larry Peterson* Timothy Roscoe† Tammo Spalink* Mike Wawrzoniak*

*Department of Computer Science, Princeton University
†Intel Research
‡Computer Science Division, University of California, Berkeley

Abstract

PlanetLab is a geographically distributed overlay network designed to support the deployment and evaluation of planetary-scale network services. Two high-level goals shape its design. First, to enable a large research community to share the infrastructure, PlanetLab provides *distributed virtualization*, whereby each service runs in an isolated slice of PlanetLab's global resources. Second, to support competition among multiple network services, PlanetLab decouples the operating system running on each node from the network-wide services that define PlanetLab, a principle referred to as *unbundled management*. This paper describes how PlanetLab realizes the goals of distributed virtualization and unbundled management, with a focus on the OS running on each node.

1 Introduction

PlanetLab is a geographically distributed overlay platform designed to support the deployment and evaluation of planetary-scale network services [30]. It currently includes over 350 machines spanning 150 sites and 20 countries. It supports over 450 research projects focused on a wide range of services, including file sharing and network-embedded storage [11, 22, 35], content distribution networks [39], routing and multicast overlays [1, 8], QoS overlays [38], scalable object location services [2, 33, 34, 37], anomaly detection mechanisms [9], and network measurement tools [36].

As a distributed system, PlanetLab is characterized by a unique set of relationships between principals—e.g., users, administrators, researchers, service providers—which make the design requirements for its operating system different from traditional hosting services or timesharing systems.

The first relationship is between PlanetLab as an organization, and the institutions that own and host PlanetLab nodes: the former has administrative control over the nodes, but local sites also need to enforce policies about how the nodes are used, and the kinds and quantity of network traffic the nodes can generate. This implies a need to share control of PlanetLab nodes.

The second relationship is between PlanetLab and its users, currently researchers evaluating and deploying planetary-scale services. Researchers must have access to the platform, which implies a distributed set of machines that must be shared in a way they will find useful. A PlanetLab "account", together with associated resources, must therefore span multiple machines. We call this abstraction a *slice*, and implement it using a technique called *distributed virtualization*.

A third relationship exists between PlanetLab and those researchers contributing to the system by designing and building *infrastructure services*, that is, services that contribute to the running of the platform as opposed to being merely applications on it. Not only must each of these services run in a slice, but PlanetLab must support multiple, parallel services with similar functions developed by different groups. We call this principle *unbundled management*, and it imposes its own requirements on the system.

Finally, PlanetLab exists in relation to the rest of the Internet. Experience shows that the experimental networking performed on PlanetLab can easily impact many external sites' intrusion detection and vulnerability scanners. This leads to requirements for policies limiting what traffic PlanetLab users can send to the rest of the Internet, and a way for concerned outside individuals to find out exactly why they are seeing unusual traffic from PlanetLab. The rest of the Internet needs to feel safe from PlanetLab.

The contribution of this paper is to describe in more detail the requirements that result from these relationships, and how PlanetLab fulfills them using a synthesis of operating systems techniques. This contribution is partly one of design because PlanetLab is a work-in-progress and only time will tell what infrastructure services will evolve to give it fuller definition. At the same time, however, this design is largely the product of our experience having hundreds of users stressing PlanetLab since the platform became operational in July 2002.

2 Requirements

This section defines distributed virtualization and unbundled management, and identifies the requirements each places on PlanetLab's design.

2.1 Distributed Virtualization

PlanetLab services and applications run in a *slice* of the platform: a set of nodes on which the service receives a fraction of each node's resources, in the form of a virtual machine (VM). Virtualization and virtual machines are, of course, well-established concepts. What is new in PlanetLab is *distributed virtualization*: the acquisition of a distributed set of VMs that are treated as a single, compound entity by the system.

To support this concept, PlanetLab must provide facilities to create a slice, initialize it with sufficient persistent state to boot the service or application in question, and bind the slice to a set of resources on each constituent node. However, much of a slice's behavior is left unspecified in the architecture. This includes exactly how a slice is created, which we discuss in the context of unbundled management, as well as the programming environment PlanetLab provides. Giving slices as much latitude as possible in defining a suitable environment means, for example, that the PlanetLab OS does not provide tunnels that connect the constituent VMs into any particular overlay configuration, but instead provides an interface that allows each service to define its own topology on top of the fully-connected Internet. Similarly, PlanetLab does not prescribe a single language or runtime system, but instead allows slices to load whatever environments or software packages they need.[1]

2.1.1 Isolating Slices

PlanetLab must isolate slices from each other, thereby maintaining the illusion that each slice spans a distributed set of private machines. The same requirement is seen in traditional operating systems, except that in PlanetLab the slice is a distributed set of VMs rather than a single process or image. Per-node resource guarantees are also required: for example, some slices run time-sensitive applications, such as network measurement services, that have soft real-time constraints reminiscent of those provided by multimedia operating systems. This means three things with respect to the PlanetLab OS:

- It must *allocate and schedule node resources* (cycles, bandwidth, memory, and storage) so that the runtime behavior of one slice on a node does not adversely affect the performance of another on the same node. Moreover, certain slices must be able to request a minimal resource level, and in return, receive (soft) real-time performance guarantees.
- It must either *partition or contextualize the available name spaces* (network addresses, file names, etc.) to prevent a slice interfering with another, or gaining access to information in another slice. In many cases, this partitioning and contextualizing must be coordinated over the set of nodes in the system.
- It must *provide a stable programming base* that cannot be manipulated by code running in one slice in a way that negatively affects another slice. In the context of a Unix- or Windows-like operating system, this means that a slice cannot be given root or system privilege.

Resource scheduling and VM isolation were recognized as important issues from the start, but the expectation was that a "best effort" solution would be sufficient for some time. Our experience, however, is that excessive loads (especially near conference deadlines) and volatile performance behavior (due to insufficient isolation) were the dominant problems in early versions of the system. The lack of isolation has also led to significant management overhead, as human intervention is required to deal with run-away processes, unbounded log files, and so on.

2.1.2 Isolating PlanetLab

The PlanetLab OS must also protect the outside world from slices. PlanetLab nodes are simply machines connected to the Internet, and as a consequence, buggy or malicious services running in slices have the potential to affect the global communications infrastructure. Due to PlanetLab's widespread nature and its goal of supporting novel network services, this impact goes far beyond the reach of an application running on any single computer. This places two requirements on the PlanetLab OS.

- It must *thoroughly account resource usage*, and make it possible to place *limits* on resource consumption so as to mitigate the damage a service can inflict on the Internet. Proper accounting is also required to isolate slices from each other. Here, we are concerned both with the node's impact on the hosting site (e.g., how much network bandwidth it consumes) and remote sites completely unaffiliated with PlanetLab (e.g., sites that might be probed from a PlanetLab node). Furthermore, both the local administrators of a PlanetLab site and PlanetLab as an organization need to collectively set these policies for a given node.
- It must make it easy to *audit resource usage*, so that *actions* (rather than just resources) can be accounted

[1]This is not strictly true, as PlanetLab currently provides a Unix API at the lowest level. Our long-term goal, however, is to decouple those aspects of the API that are unique to PlanetLab from the underlying programming environment.

```
public class Configuration {
    boolean valid;
    int sequenceNum;
    LogicalAddr primary;
    LogicalAddr[] secondary;
    String consensusID;
}
```

Figure 1: A configuration.

Om servers are grouped into configurations (Figure 1). Each configuration contains the set of servers holding copies of a particular object. A physical node may belong to multiple configurations. Conceptually, the total number of configurations equals the number of objects in Om. However, multiple objects residing on the same set of replicas share the same configuration, which significantly reduces the number of configurations and overall regeneration activity.

2.2 Two Quorum Systems for Maintaining Consistency

Throughout this paper, we use *linearizability* [18] as the definition for consistency. An *access* to an Om object is either a *read* or a *write*. Each access has a *start time*, the wall-clock time when the user submits the access, and a *finish time*, the wall-clock time when the user receives the reply. Linearizability requires that: i) each access has a unique *serialization point* that falls between its start time and finish time, and ii) the results of all accesses and the final state of the replicas are the same as if the accesses are applied sequentially by their serialization points.

To maintain consistency, Om uses two different quorum systems in two different places of the design. The first is a read-one/write-all quorum system for accessing objects on the replicas. We choose to use this quorum system to maximize the performance of read operations. In general, however, our design supports an arbitrary choice of read/write quorum. Each configuration has a *primary* replica responsible for serializing all writes and transmitting them to *secondary* replicas. The failure of any replica causes regeneration. Thus both primary and secondary replicas correspond to *gold replicas* in Pangaea [35]. It is straightforward to add additional *bronze replicas* (which are not regenerated) into our design. Distinguishing these two kinds of replicas helps to decrease the overhead of maintaining the lease graph, liveness monitoring and performing two-phase writes among the gold replicas.

Reads can be processed by any replica without interacting with other replicas. A write is always forwarded to the primary, which uses a two-phase protocol to propagate the write to all replicas (including itself). Even though two-phase protocols in WAN can incur high overhead, we limit this overhead because Om usually needs a relatively small number of replicas to achieve certain availability target [42] (given its single replica regeneration mechanism).

The second quorum system is used during reconfiguration to ensure that replicas agree on the membership of the new configuration. In wide-area settings, it is possible for two replicas to simultaneously suspect the failure of each other and to initiate regeneration. To maintain consistency, the system must ensure a unique configuration for the object at any time. Traditional approaches for guaranteeing unique configuration require each replica to coordinate with a majority before regeneration, so that no simultaneous conflicting regeneration can be initiated.

Given the availability cost of requiring a majority [42] to coordinate regeneration, we adopt the witness model [40] that achieves similar functionality as a quorum system. In the witness model, quorum intersection is not always guaranteed, but is extremely likely. In return, a quorum in the witness model can be as small as a single node. While our implementation uses the witness model, our design can trivially replace the witness model with a traditional quorum system such as majority voting.

2.3 Node Failure/Leave and Reconfiguration

The membership of a configuration changes upon the detection of node failures or explicit reconfiguration requests. Failures are detected in Om via timeouts on messages or heartbeats. By definition, accurate failure detection in an environment with potential network failure and node overload, such as the Internet, is impossible. Improving failure detection accuracy is beyond the scope of this paper.

There are two types of reconfigurations in Om: *failure-free reconfiguration* and *failure-induced reconfiguration*. Failure-free reconfiguration takes place when a set of nodes gracefully leave or join the configuration. "Gracefully" means that there are no node failures or message timeouts during the process. On the other hand, Om performs failure-induced reconfiguration when it (potentially incorrectly) detects a node failure (in either normal operation or reconfiguration).

Failure-free reconfiguration is lightweight and requires only a single round of messages from the primary to all replicas, a process even less expensive than writes. Failure-induced reconfiguration is more expensive because it uses a *consensus protocol* to enable the replicas to agree on the membership of the next configuration. The consensus protocol, in turn, relies on the second quorum system to ensure that the new configuration is unique among the replicas.

Under a denial of service (DoS) attack, all reconfigurations will become failure-induced. One concern is that an Om configuration must be sufficiently over-provisioned to handle the higher cost of failure-induced reconfiguration under the threat of such attacks. However, the reconfiguration functionality of Om actually enables it to dynamically shift to a set of more powerful replicas (or expand the replica group) under DoS attacks, making static over-provisioning unnecessary.

2.4 Node Join and Reconfiguration

New replicas are always created by the primary in the background. To achieve this without blocking normal operations, the primary replica creates a snapshot of the data and transfers the snapshot to the new replicas. During this process, new reads and writes are still accepted, with the primary logging those writes accepted after creating the snapshot. After the snapshot has been transferred, the primary will send the logged writes to the new replicas, and then initiate a failure-free reconfiguration to include them in the configuration. Since the time needed to transfer the snapshot tends to dominate the total regeneration time, Om enables online regeneration without blocking accesses.

Each node in the system maintains an incarnation counter in stable storage. Whenever a node loses its state in memory (due to a crash or reboot), it increments the incarnation number. After the node rejoins the system, it should discard all messages intended for older incarnations. This is necessary for a number of reasons: For example, otherwise a primary that crashes and then recovers immediately will not be able to keep track of the writes in the middle of the two-phase protocol.

3 Normal Case Operations

Given the overall architecture described above, we now discuss some of the complex system interactions in Om. Despite the simplicity of the read-one/write-all approach for accessing objects, failures and reconfigurations may introduce several anomalies in a naive design. Below we describe two major anomalies and our solutions.

The first anomaly arises when replicas from old configurations are slow in detecting failures, and continue servicing stale data after reconfiguration (initiated by other replicas). We address this scenario by leveraging leases [17]. In traditional client-server architectures, each client holds a lease from the server. However, since Om can regenerate from any replica, a replica needs to hold valid leases from all other replicas.

Requiring each replica to contact every other replica for a lease can incur significant communication overhead. Fortunately, it is possible for a replica to *sublease* those leases it already holds. As a result, when a replica A requests a lease from B, B will not only grant A a lease for B, it can also potentially grant A leases for other replicas (with a shorter lease expiration time, depending on how long B has been holding those leases).

Following we abstract the problem by considering replicas to be nodes in a *lease graph*. If a node A directly requests a lease from node B, we add an arc from B to A in the graph. A lease graph must be strongly connected to avoid stale reads. Furthermore, we would like the layers of recursive subleasing to be as small as possible because each layer of sublease decreases the effective duration of the lease. Define the diameter of a lease graph to be the smallest integer d, such that any node A can reach any other node B via a directed path of length at most d. In our system, we would like to limit d to 2 to ensure the effectiveness of subleasing. Overhead of lease renewal is determined by the number of arcs in the lease graph. It has been proven [16] that with $n \geq 4$ nodes, the minimal number of arcs to achieve $d = 2$ is $2(n-1)$. For $n \geq 5$, we can show that the *only* graph reaching this lower bound is a star-shaped graph. Thus, our lease graphs are all star-shaped, with every node having two arcs to and from a central node. The central node does not have to be the primary of the configuration, though it is in our implementation.

A second problem results from a read seeing a write that has not been applied to all replicas, and the write may be lost in reconfiguration. In other words, the read observes temporary, inconsistent state. To avoid this scenario, we employ a two-phase protocol for writes. In the first *prepare* round, the primary propagates the writes to the replicas. Each replica records the write in a *pending queue* and sends back an acknowledgment. After receiving all acknowledgments, the primary will start the second *commit* round by sending commits to all replicas. Upon receiving a commit, a replica applies the corresponding write to the data object. Finally, the primary sends back an acknowledgment to the user. A write becomes "stable" (applied to all replicas) when the user receives an acknowledgment. The lack of an acknowlwide-area storage system. Om logically builds upon PAST [33] and CFS [10], but achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. To the best of our knowledge, Om is the first implementation and evaluation of a wide-area peer-to-peer replication system that achieves such functionality.

Om's design targets large, infrastructure-based hosting services consisting of hundreds to thousands of sites across the Internet. We envision companies utilizing hosting infrastructure such as Akamai [2] to provide wide-area mutable data access service to users. The data may be replicated at multiple wide-area sites to improve service availability and performance. We believe that our design is also generally applicable to a broader range of applications, including: i) a totally-ordered event notification system, ii) distributed games, iii) parallel grid computing applications sharing data files, and iv) content distribution networks and utility computing environments where a federation of sites deliver read/write network services.

We adopt the following novel techniques to achieve our goal of consistent and automatic replica regeneration.

1. Traditional designs for regeneration require a *majority* of replicas to coordinate consistent regeneration. We show that by taking advantage of the *limited view divergence* property in today's Internet and by adopting the *witness model* [40], Om is able to regenerate from any single replica at the cost of a small probability of violating consistency. As a result, Om can deliver high availability with a small number of replicas, while traditional designs would significantly increase [42] the number of replicas in order to deliver the same availability. When strict consistency is desired, Om can also trivially replace the witness model with a simple majority quorum (at the cost of reduced availability) to provide strict consistency.

2. We distinguish between *failure-free* and *failure-induced* reconfiguration, enabling common reconfigurations to proceed with a single round of communication while maintaining correctness even if a failure should occur in the middle.

3. We use a *lease graph* among all replicas and a two-phase write protocol to avoid executing a consensus protocol for normal writes. Reads in Om proceed with a single round trip to any single replica, yielding the read performance of a centralized service but with better network locality.

Om assumes a crash (stopping) rather than Byzantine failure model. While this assumption makes our approach inappropriate for a certain class of services, we argue that the performance, availability, consistency, and flexible reconfiguration resulting from our approach will make our work appealing for a range of important applications.

Through WAN measurement and local area emulation, we observe that the probability of violating consistency in Om is approximately 10^{-6}, which means that on average, inconsistency occurs once every 250 years with 5 replicas and a pessimistic 12 hours replica MTTF. At the same time, the ability to regenerate from any replica enables Om to achieve high availability using a relatively small number of replicas [42] (e.g., 99.9999% using 4 replicas with node MTTF of 12 hours, regeneration time of 5 minutes and human repair time of 8 hours). Under stress tests for write throughput on PlanetLab [27], we observe that regeneration in response to replica failures only causes a 20-second service interruption.

We provide an overview of Om in the next section. The following three sections then discuss the details of normal case operations, reconfiguration, and single replica regeneration in Om. We present unsafety (probability of violating consistency) and performance evaluation in Section 6. Finally, Section 7 discusses related work and Section 8 draws our conclusions.

2 System Architecture Overview

2.1 Naming and Configurations

Om relies on Distributed Hash Tables (DHTs) [32, 38] for naming its objects. The current implementation of Om uses FreePastry [13]. Om invokes only two common peer-to-peer APIs [11] from FreePastry: **void route(key $\rightarrow$ K, msg $\rightarrow$ M, nodehandle $\rightarrow$ hint)** and **nodehandle[] replicaSet(key $\rightarrow$ K, int $\rightarrow$ max_rank)**. We use these APIs to determine the set of nodes that should hold a particular object. Om does not require any change to the FreePastry code.

DHTs do not guarantee the correctness of naming. For example, the same key may be mapped to different nodes if routing tables are stale. In Om, each node ultimately determines whether it is a replica of a certain Om object. With inconsistent routing in DHTs, user requests may be routed to the wrong node. Instead of returning an incorrect value, the node will tell the user that it does not have the data.

Consistent and Automatic Replica Regeneration

Haifeng Yu
Intel Research Pittsburgh / Carnegie Mellon University
yhf@cs.cmu.edu, http://www.cs.cmu.edu/~yhf

Amin Vahdat
University of California, San Diego
vahdat@cs.ucsd.edu, http://www.cse.ucsd.edu/~vahdat

Abstract

Reducing management costs and improving the availability of large-scale distributed systems require automatic replica *regeneration*, i.e., creating new replicas in response to replica failures. A major challenge to regeneration is maintaining consistency when the replica group changes. Doing so is particularly difficult across the wide area where failure detection is complicated by network congestion and node overload.

In this context, this paper presents Om, the first read/write peer-to-peer wide-area storage system that achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. We achieve these properties through the following techniques. First, by utilizing the *limited view divergence* property in today's Internet and by adopting the *witness model*, Om is able to regenerate from any single replica rather than requiring a majority quorum, at the cost of a small (10^{-6} in our experiments) probability of violating consistency. As a result, Om can deliver high availability with a small number of replicas, while traditional designs would significantly increase the number of replicas. Next, we distinguish *failure-free* reconfigurations from *failure-induced* ones, enabling common reconfigurations to proceed with a single round of communication. Finally, we use a *lease graph* among the replicas and a two-phase write protocol to optimize for reads, and reads in Om can be processed by any single replica. Experiments on PlanetLab show that consistent regeneration in Om completes in approximately 20 seconds.

1 Introduction

Replication has long been used for masking individual node failures and for load balancing. Traditionally, the set of replicas is fixed, requiring human intervention to repair failed replicas. Such intervention can be on the critical path for delivering target levels of performance and availability. Further, the cost of maintenance now dominates the total cost of hardware ownership, making it increasingly important to reduce such human intervention. It is thus desirable for the system to automatically *regenerate* upon replica failures by creating new replicas on alternate nodes. Doing so not only reduces maintenance cost, but also improves availability because regeneration time is typically much shorter than human repair time.

Motivated by these observations, automatic replica regeneration and reconfiguration (i.e., change of replica group membership) have been extensively studied in cluster-base Internet services [12, 34]. Similarly, automatic regeneration has become a necessity in emerging large-scale distributed systems [1, 10, 20, 25, 30, 33, 35]. One of the major challenges to automatic regeneration is maintaining consistency when the composition of the replica group changes. Doing so is particularly difficult across the wide-area where failure detection is complicated by network congestion and node overload. For example, two replicas may simultaneously suspect the failure of each other, form two new disjoint replica groups, and independently accept conflicting updates.

The focus of this work is to enable automatic regeneration for replicated wide-area services that require some level of consistency guarantees. Previous work on replica regeneration either assumes read-only data and avoids the consistency problem (e.g., CFS [10] and PAST [33]), or simply enforces consistency in a best-effort manner (e.g., Inktomi [12], Porcupine [34], Ivy [25] and Pangaea [35]). Among those replication systems [1, 6, 20, 30, 37] that do provide strong consistency guarantees, Farsite [1] does not implement replica group reconfiguration. Oceanstore [20, 30] mentions automatic reconfiguration as a goal but does not detail its approach, design or implementation. Proactive recovery [6] enables the same replica to leave the replica group and later re-join, but still assumes a fixed set of replicas. Finally, replicated state-machine research [37] typically also assumes a static set of replicas.

In this context, we present Om, a read/write peer-to-peer

[4] ALIGNMENT SOFTWARE. AppAssure, 2002. http://www.alignmentsoftware.com.

[5] AMMONS, G., AND LARUS, J. R. Improving Data-flow Analysis with Path Profiles. In *Conference on Programming Language Design and Implementation* (1998), pp. 72–84.

[6] AT&T LABS. Graphviz, 1996. http://www.research.att.com/sw/tools/graphviz.

[7] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Real-time Modelling and Performance-aware Ssytems. In *HotOS IX* (May 2003).

[8] BEA SYSTEMS. WebLogic. http://www.bea.com.

[9] BREIMAN, L., H.FRIEDMAN, J., OLSHEN, R. A., AND STONE, C. J. *Classification and Regression Trees*. Wadsworth, 1984.

[10] BREWER, E. Lessons from Giant-Scale Services. *IEEE Internet Computing 5*, 4 (July 2001), 46–55.

[11] C. AMZA ET AL. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization* (Nov 2002).

[12] CHEN, M., KICIMAN, E., ACCARDI, A., FOX, A., AND BREWER, E. Using Runtime Paths for Macroanalysis. In *HotOS IV* (2003).

[13] CHEN, M., KICIMAN, E., FRATKIN, E., BREWER, E., AND FOX, A. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Symposium on Dependable Networks and Systems (IPDS Track)* (2002).

[14] D. PATTERSON ET AL. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. CSD-02-1175, UC Berkeley Computer Science, 2002.

[15] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI* (2002).

[16] ELSON, J., GIROD, L., AND ESTRIN, D. Fine-Grained Network Time Synchronization Using Reference Broadcasts. In *OSDI* (2002).

[17] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI* (2000).

[18] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-Based Scalable Network Services. In *SOSP* (1997), pp. 78–91.

[19] FOX, A., AND PATTERSON, D. When Does Fast Recovery Trump High Reliability? In *Workshop on Evaluating and Architecting System dependabilitY (EASY)* (October 2002).

[20] FRANCISCO, C. A., AND FULLER, W. A. Quantile Estimation with a Complex Survey Design. *The Annals of Statistics 19*, 1 (1991), 454–469.

[21] GNU. Octave, 1992. http://www.octave.org.

[22] GRAHAM, S., KESSLER, P., AND MCKUSICK, M. gprof: A Call Graph Execution Profiler. In *Symposium on Compiler Construction* (June 1982), vol. 17, pp. 120–126.

[23] GRAY, J. Dependability in the Internet Era. http://research.microsoft.com/~gray/talks/InternetAvailability.ppt.

[24] HELLERSTEIN, J. L., MACCABEE, M., MILLS, W. N., AND TUREK, J. J. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *International Conference on Distributed Computing Systems* (1999).

[25] HELLERSTEIN, J. L., ZHANG, F., AND SHAHABUDDIN, P. An Approach to Predictive Detection for Service Management. In *Symposium on Integrated Network Management* (1999).

[26] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, third ed. Morgan Kaufmann, 2002. Chapter 8.12.

[27] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[28] IDENTIFY SOFTWARE. AppSight, 2001. http://www.identify.com.

[29] J. SRIVASTAVA ET AL. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. *SIGKDD Explorations 1*, 2 (2000), 12–23.

[30] JBOSS.ORG. J2EE App Server. http://www.jboss.org.

[31] LARUS, J. R. Whole Program Paths. In *Conference on Programming Languages Design and Implementation* (May 1999).

[32] LEE, W., AND STOLFO, S. Data Mining Approaches for Intrusion Detection. In *USENIX Security Symposium* (1998).

[33] M. MEIER ET AL. Experiences with Building Distributed Debuggers. In *SIGMETRICS Symposium on Parallel and Distributed Tools* (1996).

[34] MANNING, C. D., AND SHUTZE, H. *Foundations of Statistical Natural Language Processing*. The MIT Press, 2000.

[35] MICROSOFT. .NET. http://microsoft.com/net.

[36] MICROSOFT RESEARCH. Magpie project, 2003. http://research.microsoft.com/projects/magpie.

[37] MILLS, D. L. RFC 1305: Network time protocol (version 3) specification, implementation, Mar. 1992.

[38] MOSBERGER, D., AND PETERSON, L. L. Making Paths Explicit in the Scout Operating System. In *OSDI* (1996).

[39] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *USITS* (March 2003).

[40] ORACLE. Oracle Data Mining, 2002. http://technet.oracle.com/products/bi/9idmining.html.

[41] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks 31*, 23–24 (1999), 2435–2463.

[42] RATIONAL SOFTWARE. Quantify for UNIX. http://www.rational.com/products/quantify_unix.

[43] REUMANN, J., MEHRA, A., SHIN, K. G., AND KANDLUR, D. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference* (June 2000), pp. 117–130.

[44] RICE, J. A. *Mathematical Statistics and Data Analysis*, second ed. Duxbury Press, 1994.

[45] S. D. GRIBBLE ET AL. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks 35*, 4 (2001), 473-497.

[46] SITRAKA. PerformaSure, 2002. http://www.sitraka.com/software/performasure.

[47] SUN MICROSYSTEMS. Java2 Enterprise Edition (J2EE). http://www.javasoft.com/j2ee.

[48] TEALEAF TECHNOLOGY. IntegriTea, 2002. http://www.tealeaf.com.

[49] THOMAS GSCHWIND ET AL. WebMon: A Performance Profiler for Web Transactions. In *4th IEEE Int'l Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems* (2002).

[50] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC-W Benchmark Specification, Web-based Ordering. http://www.tpc.org/wspec.html.

[51] WAGNER, D., AND DEAN, D. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy* (2001), pp. 156–169.

[52] WARD, A., GLYNN, P., AND RICHARDSON, K. Internet Service Performance Failure Detection. In *Web Server Performance Workshop* (1998).

[53] WEISER, M. Program Slicing. *IEEE Transactions on Software Engineering SE-10*, 4 (1984), 352–357.

[54] WELSH, M., CULLER, D. E., AND BREWER, E. A. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP* (2001), pp. 230–243.

[55] WORLD WIDE WEB CONSORTIUM (W3C). Voice Extensible Markup Language (VoiceXML) Version 2.0. http://www.w3.org/TR/voicexml20.

[56] YEMINI, A., AND KLIGER, S. High Speed and Robust Event Correlation. *IEEE Communication Magazine 34*, 5 (May 1996), 82–90.

the dominant causal paths through a distributed system of black boxes using message-level traces, without any knowledge of node internals or message semantics. They have used the causal paths to profile and diagnose performance problems. Although their approach is less invasive than ours, it is more difficult to associate specific observations with specific request paths for failure management tasks.

7.2 Paths in Other Contexts

Whole Program Paths [31] shares our view of capturing program dynamic control flow, and applies this to detect hot sub-paths in individual processes for performance tuning. Scout [38], Ninja [45], SEDA [54], and Virtual Services [43] use paths to specify control flow and resource allocation while servicing requests. These systems would be particularly easy to integrate with our macro analysis infrastructure.

Clickstream analysis uses paths to represent the sequence of web pages a user visits. They focus on understanding user behavior to improve usability, characterize visitors, and predict future access patterns [29].

7.3 Diagnosis

IntegriTea [48] and AppSight [28] trace requests to record method calls and parameters to capture and replay single-path failures. These systems do not correlate across multiple requests for improved diagnosis. Some distributed debuggers support stepping through remote calls [33]. These tools typically work with homogeneous components and aid in low-level debugging.

Event and alarm correlation techniques have been used for many years in network management applications [56]. The challenge is to infer event causality from just the event sources and timestamps.

7.4 Anomaly Detection

Anomaly detection techniques have been used to identify software bugs [17, 51] and detect intrusions [32] from resource usage [15], system calls [27], and network packets [41]. Performance failure detection tools [25, 52] compensate for workload variations using trend analysis. The workload models are constructed per site or per request type. Paths enable fine-grain workload models to be constructed per component and per request type.

8 Conclusion

We have presented a new approach to managing failures and evolution using paths. We trace request paths through distributed system components, aggregate the resulting data, and apply statistical techniques to infer high-level system properties.

This approach allows us to build tools that apply to many applications, even including legacy applications that run on application servers. These tools can perform complex detection and diagnosis operations without detailed knowledge of the applications or the system components. The key to this ability is a large amount of traffic (many paths), which enables meaningful automated statistical analysis.

We have applied our methodology to address four key challenges that arise when managing large, distributed systems.

1. Path anomalies and latency profiles can be used to quickly and accurately detect system failures, including both correctness and performance problems. Because of automated statistical methods, we can run fault detectors for many different kinds of failures simultaneously.
2. Paths help isolate faults and diagnose failures, either for defects in a single path or sets of paths that exhibit some property.
3. Paths allow us to estimate the impact of a fault by finding the set of other paths that exhibit similar behavior, which allows for prioritization of problems and resolution of SLA deviations.
4. Paths contribute to correct system evolution by discovering system structure and detecting subtle behavioral differences across system versions. As with detection, automated tools allows us to track thousands of system metrics simultaneously, looking for statistically significant deviations from the old version.

We have validated our methodology using three implementations: a research prototype and two large, commercial systems that service millions of requests per day.

We are currently exploring a variety of data storage and analysis engine architectures in order to build a path-based infrastructure whose use can scale far past the systems it supports. We also continue to experiment with new algorithms for our analysis engines. We are exploring the application of our failure detection techniques to clickstream paths. Finally, we plan to extend paths to peer-to-peer systems, where we can use path properties to verify distributed macro invariants so as to detect faulty implementations or misbehaving members.

References

[1] ABBOTT, M. B., AND PETERSON, L. L. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking 1*, 5 (1993), 600–610.

[2] AGRAWAL, R., IMIELINSKI, T., AND SWAMI, A. Mining association rules between sets of items in large databases. In *SIGMOD* (1993), pp. 26–28.

[3] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP* (2003).

applications and Tellme runs hundreds of voice applications that are all observed and analyzed with zero attention required of the application developers.

Large-scale system design is moving toward the application server model [8, 35, 45], where the platform acts as a cluster-based operating system that provides resource management, scheduling, and other services for applications. We believe that the path abstraction should be a core design principle inherent in these systems.

We offer several design guidelines for building path-based analysis support into a system. The first is to track path request state, including unique path IDs, as close to the control flow in the code as possible. Develop interfaces, classes, and messaging protocols with this in mind, rather than addressing versioning issues later.

The second guideline is to use a high-throughput logging mechanism and a space-efficient data representation, so that the system's critical path is minimally impacted. Although text logs are easier to parse and understand, we have found that compressed binary representations significantly reduce logging overhead.

The third is to design the query interface not only with the expected formal use in mind, but to also plan for exploratory data analysis. Engineers tend to form new questions when working with path data, and the interface must be general enough to support such experimentation.

The fourth guideline is to design efficient, flexible query and aggregation sub-systems. Centralized versus distributed storage and processing tradeoffs must be weighed based on expected use – a distributed design leverages parallel I/O and processing, but adds complexity and runtime overhead to each node in the system. We stress that *data is cheap* in large systems. A lot of data goes a long way with simple, inexpensive statistical tests. These simple tests also tend to scale better.

6.4 Distributed Timestamps

Latency distribution analysis is challenging when a path runs through machines with different clocks and the intervals of interest are short. We recommend using local timestamps (e.g., Solaris `gethrtime`) when possible to avoid clock drift concerns, and to use system timestamps (e.g., `gettimeofday`) with a time-synchronization protocol such as NTP [37] to compute intervals between hosts on different clocks. ObsLogs contain a system timestamp per path per machine, and all observations are made with 32-bit local timestamps. So far the extra noise of inter-machine intervals has not been a limitation in practice, although there are improved time synchronization algorithms that we could deploy to reduce the noise [16].

6.5 Path Limitations

The path-based approach has proven useful in many applications. However, the path concept itself does not solve problems. Rather, paths provide a framework by which existing techniques may be better focused to tackle common challenges encountered while managing large, distributed systems.

We must decide which observations to include in paths for the system of interest. Software bugs may be so focused that they slip between two observations, so that their impact is not noticed but for the coarse path detail. Deciding what to instrument is an iterative process that evolves with the system. We prefer to err on the side of excess, and dynamically filter out noisy observations that are not as frequently useful.

For example, our current Pinpoint Tracer implementations generate observations at the application level but do not yet account for lower-level components, such as transparent web caches and RAID disks. Extending Tracers to include such information can help us detect and diagnose low-level configuration errors and failures (e.g., buggy RAID controllers).

7 Related Work

In this section we discuss previous work in path-based profiling, paths, diagnosis, and anomaly detection.

7.1 Path-based Profiling

There are many profiling tools that provide more complete solutions for specialized tasks, such as performance modeling. None of these systems are applicable to the failure management tasks discussed in this paper except for performance failures.

Similar to our approach, several systems use explicit identifiers to trace requests through multi-tier systems for performance profiling. Some also implement the tracing in the platform so that no application modification is required. Magpie [7] profiles distributed systems to observe processing state machines (e.g., for HTTP requests) and to measure request resource consumption (CPU, disk, and network usage) at each stage. Magpie then builds stochastic workload models suitable for performance prediction, tuning, and diagnosis. WebMon [49] uses HTTP cookies as identifiers to trace requests from web browsers to the web servers and application servers. There are also several recent commercial request tracing systems focusing on performance profiling and diagnosis, such as PerformaSure [46] and AppAssure [4]. ETE [24] is a customizable event collection system for transactions. It requires manually written transaction definitions to associate events with a transaction, whereas paths captures the association automatically.

Aguilera *et al.* [3] takes a completely non-invasive approach to profiling distributed systems. They infer differences.[4] In fact, Tellme conducts such tests on all its data to automate the analysis process, so a human is only required when a significant behavior change is detected. The p-value for the Mann-Whitney test for versions 1 and 2 is 1.5×10^{-4}, so we have high confidence that median search time increased in version 2.[5] However, taking a different look at this data provides more insight.

At the right of Figure 8, we focus our attention on the search time outliers by using sample survivor functions on a logarithmic scale. The aberrant version 2 behavior is now clear: all quantiles above 80% have increased.

The culprit became evident after using the path information to identify the responsible subinterval: application F's data feed changed in version 2, and we were witnessing erratic web server performance.

In summary, paths offer powerful tools for evolving systems. Automatically deriving system structure and dependencies helps development, QA, and operations teams better understand the system. We can also automatically detect statistically significant changes in performance and perceived latency across versions. Some differences are only visible in distribution outliers, and not in the mean, but they are still captured by this approach. We may further employ paths to understand the cause behind each change, per Section 4.2.

6 Discussion

In this section, we summarize some important lessons that we have learned while working with paths.

6.1 Maintainability and Extensibility

For Tracers to be practical, the instrumentation must be: 1) maintainable, so that it is not likely to break in light of ongoing platform changes, and 2) extensible, so new platform additions can easily make use of the measurement infrastructure. Path-based instrumentation succeeds in this regard because it keeps the reporting and analysis logic external to the instrumented system.

Consider the common problem of instrumenting software to measure the latency of a certain interval, that begins in one software module and ends in another. These modules know little of each other's internals.

The naive *point-based* approach would have the two modules log the start and end events respectively, and *attempt* to pair them up externally. Without a path to link these two endpoints to the same request, it is easy to get confused about the pairings. Although it is possible to statistically infer the common pairings [3], we are often interested in precise matches.

[4]We prefer tests that avoid assumptions about the distributions (non-parametric tests). Here we use the Mann-Whitney test to determine if two datasets come from the same distribution. For testing changes in quantiles, we prefer Woodruff confidence intervals [20].

[5]We say differences are statistically significant only for low p-values (< 0.001), as we have a great deal of data at our disposal.

We could internalize (but not eliminate) the pairing task by explicitly tracking and propagating timestamps inside the system. This *interval-based* approach offers limited flexibility for adding and modifying measurements, and adds intricate, and often overlooked, dependencies between modules.

The *path-based* approach provides the flexibility without the dependencies. The modules simply report observations with timestamps, and paths ensure a correct pairing of these two events. In addition, measurements of new internals (sub-paths) can be easily added without new instrumentation.

At Tellme, we have repeatedly watched measurements made using older approaches break, while path-based measurements have remained robust in the face of rapid change. With path-based measurement, developers estimate that they now spend 23-28 fewer person-hours per new, medium-sized software component validating correctness and overall performance impact.

6.2 Trapping Failures

Given the difficulty of perfectly simulating real workloads during testing, our philosophy is to accept the fact that failures will happen, but to be well prepared to recover quickly [14]. The sheer size and large request rate of a full production system expose rare behavior more frequently than offline tests. We sometimes cannot afford to wait for the problem to recur, and attempts to reproduce subtle timing conditions on a smaller test system can be difficult.

Paths allow us to isolate a problem the first time it happens, and if we cannot immediately diagnose the problem, we can identify the areas where we need to enable additional instrumentation to catch the problem the next time. Tellme has used this *failure trapping* approach to successfully diagnose rare and obscure bugs. We use the dynamic filter described in Section 3.2 to enable the aggregation of verbose observations and trap problems in this manner.

This is one application area where paths aid in data mutation detection. If content is a key part of the critical path of a system, such as audio in Tellme's network, it is advantageous to record data signatures (e.g., hashes) as observations at various points along a path, so that the content's integrity can be validated throughout. If it is not obvious *a priori* what data to validate, or validation generates too much data to handle during normal operation, then data validation is best enabled when attempting to trap an existing, known failure.

6.3 Designing for Path-Based Macro Analysis

Our black-box path observation approach provides visibility into application behavior and structure without requiring any application changes. Pinpoint has four

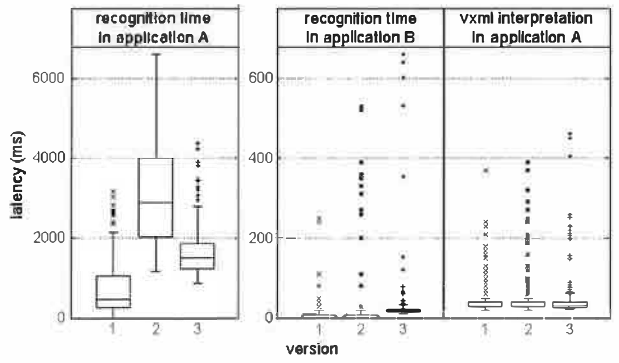


Figure 6: A trend specific to recognition time in Tellme application A suggests a regression in a speech grammar in that application.[3] Recall that the Tukey boxplots shown illustrate a distribution's center, spread, and asymmetries by using rectangles to show the upper and lower quartiles and the median, and explicitly plotting each outlier [44].

carefully controlled test environment. We analyze the resulting paths and compare them with similar data for previous platform and application versions, in order to identify regressions as well as validate performance improvements. We run 10-15 performance tests per release, each of which involves hundreds of distinct sub-path comparisons. Although we focus on latency data here, it is straightforward to apply these techniques to other resource usage and performance metrics.

The path-based approach is particularly appealing to a QA team, as many meaningful comparisons may be derived from the embedded sub-paths. This allows a QA engineer to approach the analysis task with a different perspective than the developer, and as a result, QA often identifies interesting, unanticipated path structures.

For simplicity, we consider a sample of three such tests, using three different interval types from six different applications. *Search time* is a user-perceived latency, defined as the time from when a user stops speaking to when he hears audio in response. This consists of several disjoint subintervals. *Recognition time* covers up to speech recognition completion, at which point the platform conjectures what was said. *VXML interpretation* follows, and represents the time spent evaluating the application content in response to the user's input. The final portion of search time is spent preparing the desired prompt for playback.

In our first test, shown in Figure 6, we see that recognition time in application A changed drastically in version 2. However, recognition time in all other applications for this test remained steady (as exemplified by the plot for application B in the middle), and other intervals in application A, such as VXML interpretation time, did not change either (shown on the right). This narrows the problem down considerably.

An application history check revealed a feature addition in version 2, that wound up consuming more resources than desired. A more efficient alternative was subsequently implemented in version 3.[3]

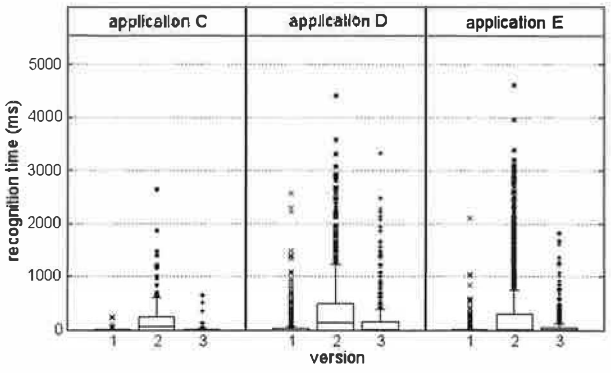


Figure 7: Consistent trends in Tellme recognition time profiles across applications suggest systemic changes.

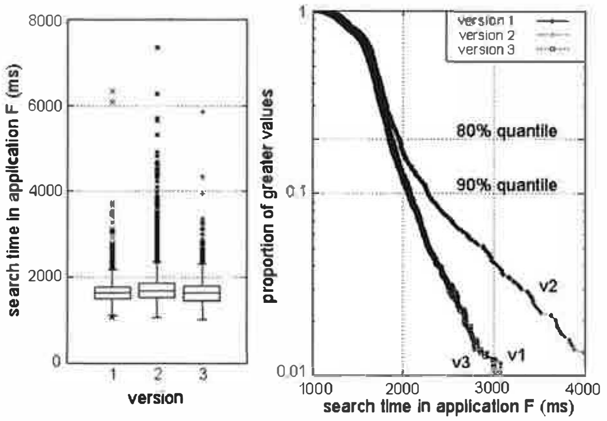


Figure 8: The regression in Tellme application F is difficult to discern in the boxplot, but is easily visible in the logarithmic survivor plot. It is also detectable quantitatively using a statistical two-sample test, and diagnosed with sub-path information.

A variety of other regressions can be identified in similar ways. Recognition times for 3 applications are shown in Figure 7. What is normally an extremely quick response takes a significant amount of time in version 2. Furthermore, this behavior was evident in *all* recognition times for this particular stress test, for *all* applications. These facts point to a systemic problem with the recognition subsystem, and indeed, this test was conducted with different recognition hardware. The latencies are all reduced in version 3, running on the usual hardware.

Our last application reveals a more subtle regression in Figure 8. The version 2 distribution appears slightly extended, although it is not visually apparent from the boxplot how statistically significant this difference is. We use standard two-sample tests [44] to quantify such

[3]Notice two other changes in the version 3 boxplots in Figures 6 and 7. First, all recognitions take slightly longer, because we upgraded the acoustic models used. Second, measurement timing resolution has improved from 10 ms, due to an operating system change.

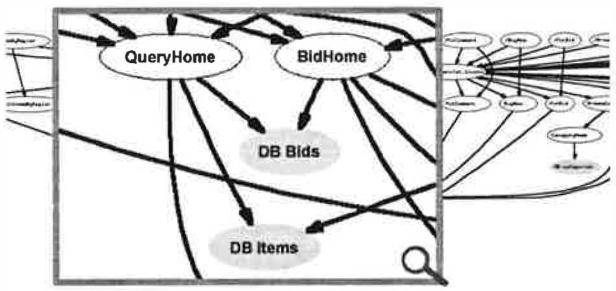


Figure 5: A portion of the derived application structure for RUBiS (a J2EE benchmark [11]) showing a subset of its 33 components, including JSPs, Servlets, EJBs, and database tables (in gray). The nodes are requests, components, and database tables. The directed edges represent observed paths.

when presented with a simulated input load (call traffic) that stresses the system past its normal operating point. Given the large amount of traffic and long time frames over which the correct behavior was observed, we can empirically bound the probability of a false alarm to less than 1×10^{-6}. Now, after little work, we had an accurate understanding of how this problem would impact live users, and could appropriately prioritize fixing it.

Note that this information also allows us to craft a targeted monitor for future occurrences of this problem, an example of failure management feedback. We can also generalize our queries to capture a larger class of problems by focusing on the user behavior. For example, we can search for callers who hang up on an incomplete path after a long period of inactivity.

5 Evolution

Systems change over time, as software updates fix bugs and introduce new bugs and features, and hardware additions increase capacity as well as improve performance. Paths address system evolution in two ways:

- **Paths capture system structure** and component dependencies, and can be used to observe systems without *a priori* knowledge of their inner workings.
- When applied to testing during a system release process, paths enable developers and Quality Assurance (QA) engineers to quickly and accurately **validate new behavior** and **identify subtle regressions** across system versions. This can be done efficiently, so that a single test run may simultaneously detect and diagnose multiple regressions.

5.1 Deducing Application Structure

Modern applications tend to have intricate, dynamic structures containing components with complex dependencies. An accurate view of such structure increases code accessibility, enables more efficient development, aids in system testing, and increases the visibility of configuration failures.

	Database Tables				
Request Type	Product	Signon	Account	Banner	Inventory
verifysignin	R	R	R		
cart	R			R	R/W
commitorder	R				W
search	R			R	
productdetails	R				R/W
checkout					W

Table 3: **An automatically generated partial state dependency table for Pet Store.** To determine which request types share state, group the rows by common entry under the desired column. For example, the `checkout` request only writes to the Inventory table, which is shared with three other requests: `cart`, `commitorder`, and `productdetails`.

Current techniques for understanding application structure rely on static analysis or human diligence to document and track application changes, sometimes with the aid of a modeling language, such as UML. Manually tracking changes is time consuming, prone to error, and difficult to enforce consistently. Paths can be used to discover this structure and provide more accurate details. With instrumented application servers, it is possible to deduce application structure without any knowledge of the application.

Note a key distinction between our approach and static analysis: paths capture *actual*, observed component dependencies, including runtime resources, instead of *potential* component dependencies. Figure 5 shows an example of an automatically derived application structure. We ran an unmodified J2EE benchmark, RUBiS[11], hosted on Pinpoint, and generated workload using the client emulator from the RUBiS package. The observed paths are aggregated to show the dependency between the various application components, including Java Server Pages, Servlets, Enterprise Java Beans, and database tables.

Paths can also be used to derive more complex application structure. For example, a database table containing end-user information is typically read and modified by several components, including those for register, login, and update operations. A bug in one of these operations may cause the others to fail. Table 3 is an automatically derived state-dependency table for an unmodified Pet Store application, showing the actual database tables read and written by various requests. Such knowledge is useful when diagnosing state-dependent bugs or data corruption, and understanding inter-path dependencies.

5.2 Versioning

Identifying differences in system performance and behavior is an important part of upgrading applications and the platform they run on. During QA testing at Tellme, we use a telephone call simulator to stress systems in a components so that individual component logs yield a diagnosis. In the empty audio clip case at Tellme (see Section 4.1.1), an engineer familiar with a particular application noticed a short audio playback and provided the timestamp and caller ID of a phone call that enabled us to quickly locate the relevant path. Once we had visualized the latency profile, a short 20 ms playback time suggested an empty audio clip. The preceding observations confirmed that a remote audio server thought it had successfully serviced the audio request, when in fact a rare error had occurred. We identified the particular remote process from the path information, and text logs on that machine subsequently revealed the root cause.

4.2.2 Multi-path Diagnosis

Statistical techniques help rapidly narrow down potential causes of failure by correlating data across multiple paths [13]. This works with black-box components, and can be done automatically and independently of system upgrades. The heavy traffic presented to a large, live system exposes rare behavior and strengthens all our statistical tools by narrowing confidence intervals. In many systems, it is cost-prohibitive to reproduce such large, realistic input loads offline. For the rest, these techniques prove equally powerful offline.

To isolate faults to the components responsible, we search for correlations between component use and failed requests. Such components frequently cause the failures, or at least provide more insight into the chain of events leading to the failures. This can be cast as a feature selection problem in the statistical learning domain, where the features are the components and resources used by each request.

We have explored both a statistical machine learning approach and a data mining approach. The former involves training a decision tree [9] to differentiate between classes of success and failure, where the tree edges leading to failures become root cause candidates. The latter approach, called association rules [2], uses brute force to formulate all feature combinations (the rules) that are observed to co-occur with failures, and then ranks these by the conditional probability of failure.

We used two heuristics in implementing these algorithms. The first is noise filtering. Because large systems rarely operate with perfect consistency, they usually exhibit minor but acceptable abnormal behavior. When diagnosing failures, we are interested in identifying the root causes that result in a large portion of overall abnormal behavior. Therefore, we discard those root causes that fail to explain a sizable portion. The second heuristic trims the set of identified causes by eliminating redundant features that correlate perfectly with each other.

To evaluate these approaches, we collected 10 failure traces from eBay's production site. Four traces had two independent failures each for a total of 14 root causes, consisting of machine, software, and database faults. Each trace had roughly 50 features that tracked 260 machines, 300 request types, 7 software versions, and 40 databases. Both techniques correctly identified 93% of the root causes. The decision tree produced 23% false positives, compared with 50% for the association rules.

We have deployed an online diagnosis tool based on a greedy variant of the decision tree approach to eBay's production site. Instead of building a full decision tree, the algorithm locates the single fault that results in the largest number of failures. The tool analyzes 200K paths in less than 3 seconds, and sends diagnosis results via real-time Tivoli alerts to the operations team.

4.3 Impact Analysis

Failure impact on a system's users is a key component of many service quality metrics. For billing purposes, it is important to measure such impact accurately. It is also important to measure it *quickly*, so that problem solving resources may be correctly prioritized.

We measure the proportion of requests that are successfully serviced, as defined in a Service Level Agreement (SLA). A thorough SLA takes high-level, end-user factors into consideration, such as the quality of various responses or of the entire session. These are richer service metrics than availability.

Paths provide a means to accurately and rapidly compute such metrics. This is similar to the detection problem, but different in that we are not satisfied with just knowing whether a problem is happening, but rather want to identify *every* request impacted by the root cause, regardless of the different possible manifestations.

Using the details in the path collision example, we were able to perform an accurate impact analysis for a stress test, where we predict how often such race conditions would occur and be user-visible in a production environment. This is *predictive* impact analysis, because we are using results in an offline test environment to predict potential outcomes in a production environment. We can also perform *retroactive* impact analysis, where as part of a failure postmortem, paths help us answer a different set of questions, including how particular applications are impacted and to what degree.

For example, we queried for incoming call paths that stalled at the last recorded observation in Figure 3 for at least 100 ms before being interrupted by a call disconnect. All phone calls experiencing this problem stall at the same place, where the only input the user can provide is to hang up the phone. Therefore, our query would not miss any callers that experience the problem's symptoms. Our experience with the working, production version of this system indicates that this last observation is always followed by another within 10 ms, even portal, could accept many more combinations of personalization features than it had actually observed in its training set. This also means the PCFG might generate false negatives, allowing bad paths to slip through. Though this false-negative effect has not been a major factor in our experimental results, a context-sensitive grammar would make a different tradeoff, allowing fewer false-negatives but likely more false-positives.

Once deployed, we single out rare paths, as determined by our trained PCFG. In the results presented here, we only consider a path's structure to be anomalous if it does not fit our PCFG at all.

We implemented and evaluated this algorithm in Pinpoint, testing it by injecting a series of failures into two implementations of Sun's sample e-commerce site for J2EE, a clustered Petstore v1.1.2 and a rearchitected single-node Petstore 1.3.1.

We modified the J2EE platform to allow us to inject various failures, including expected and unexpected exceptions, as well as omitted method calls. In our experiments, we injected in turn each of the three kinds of failures into each of the 24 EJBs in the Petstores. For each failure experiment, we stressed the system with an application-specific workload for 5 minutes. We used our own trace-based load generator for each site, with a workload mix approximating that of the TPC web e-commerce ordering benchmark, TPC-W WIPSo [50].

We first trained Pinpoint's PCFG model with the paths from a fault-free run of the application under a 5 minute workload. Then, we used this model to score the paths observed during each experiment. Table 2 summarizes our results. We successfully detected 90% of all the failures. Overall, structural anomaly tests performed as well as or better than simple HTTP error and log monitoring. Also, HTTP monitoring found almost exclusively secondary faults, without noticing requests directly injected with faults. In comparison, structural anomaly detection correctly identified both types of faulty requests.

Although our path structure anomaly detection implementation excelled in these experiments, there are a number of realistic scenarios where it would not fare as well. During a software upgrade, an application's behavior may change significantly, and the PCFG may require explicit retraining. Also, some failures may have a subtle impact on path structure, so that the critical details are not instrumented as observations. For example, a data manipulation bug may not impact the associated path structure or the latencies recorded.

4.1.3 Latency Anomalies

Paths allow us to construct performance models for components that vary by request type, such as URLs. Such modeling details are valuable, since many components behave very differently when presented with different inputs. For example, a `UserProfile` component may service `getProfile` requests more quickly than `updateProfile` requests.

Deviations in system latencies often signal problems. An increase in latency outliers may indicate partial failures, while an increase in average latency might suggest overload. Latency decreases may be caused by errors preventing a request from being fully serviced.

In an empty audio clip example at Tellme, an audio server experienced an internal fault, so that it produced much shorter audio playbacks than the desired ones. This failure is not catastrophic at the system level; a rare, short audio playback typically goes unnoticed by the user. The problem is therefore difficult to detect via low-level system monitoring and would otherwise require significant application knowledge to handle effectively.

Despite this challenge, we were able to query the ObsLogs to detect all similar occurrences once we understood the path latency characteristics for this failure: observations for the beginning and end of playback spaced too closely together in time. We correlated across multiple failure paths to deduce which components affected which applications, so we could isolate the failing components and assess application impact. With this new knowledge, we crafted a monitor to use these sub-path latency deviations to detect any future failures in both our production and testing environments; this is an example of the feedback aspect of failure management.

4.2 Diagnosis

Paths aid in diagnosis by identifying the components involved and linking the symptoms with the distributed state responsible. Although a single path may be sufficient to guide developers to identify the root causes, multiple paths enable the use of statistical techniques to build automated diagnosis tools.

We stress that this process involves system data from when the problem is first witnessed. We treat offline failure reproduction as a backup plan for when we do not have enough information to determine the root cause.

4.2.1 Single-path Diagnosis

A single path, when sufficiently instrumented with details such as function arguments and database queries, is sufficient for debugging many software problems. The control flow embodied by paths guides the use of local analysis tools, such as application-level logs, to discover and associate component details with a particular request. Without paths, the individual component logs are less useful for lack of an association between log entries and other system state for common requests. Correlating traditional debug logs for a large number of different components is painful and prone to error.

A path's latency profile may help isolate problematic

4.1 Failure Detection

Traditional monitoring methods use either low-level mechanisms, such as pings and heartbeats, or high-level application functionality tests. These low-level methods work well for fail-stop faults, but do not detect more subtle application-level failures. On the other hand, end-user tests such as web page content validation can detect broken applications, but the validation logic must be maintained and versioned along with the applications. Many services do not use these techniques because of the development and maintenance costs involved [39]. Neither approach works well for more subtle behaviors that affect performance or a small portion of requests.

Our approach is to characterize distributions for normal paths and then look for statistically significant deviations to detect failures. This can be applied to structural changes in paths. For example, an error handler could cut a path short, and a non-responsive or sluggish component could stall a path. This approach can also be applied to performance anomalies, which are visible as changes in latency distributions.

4.1.1 Path Collisions

While servicing a user request, a system may receive a second, new request that effectively aborts the first. For example, users frequently interrupt HTTP requests either by reloading the page (reissuing the request), or by clicking on a new link before the current request completes. Similarly, for Tellme applications, users may abort by hanging up, or issue new requests via touch-tone commands. We wish to tie these independent but related requests together to better understand their interaction.

The first path was interrupted and never completed. Incomplete paths often indicate some of the most challenging problems, as they capture scenarios in which the user aborts a request (or hang ups) on the system before receiving a response. For example, stalled system components often fail to exhibit faulty behavior directly, but instead show up via an increase in aborted requests.

Because of their importance, we wish to retain more context for incomplete paths. An incomplete, aborted path references the completed path that interrupted it, so we have a broader picture of how the user and other system components react to the potential problem. This is accomplished by the code that processes the abort, which associates the corresponding paths using path identifiers, memory pointers, etc. Then to find path collisions, we need only look for paths that are linked in this manner.

We consider a Tellme path collision problem in Figure 3. This was caused by a race condition that resulted in a test caller hearing complete silence, or "dead air". The paths for such stalled calls provide a clear diagnosis. The last observation recorded during the incoming call path was made just 453 ms after receiving the call. An

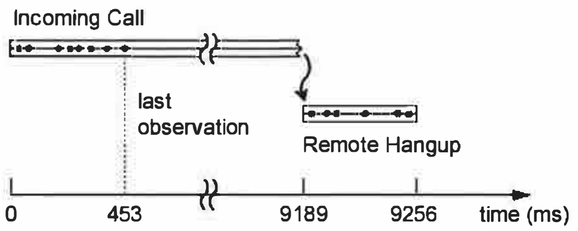


Figure 3: A stalled and subsequently interrupted Tellme response path, where an engineer hangs up after hearing dead air during a test.

Fault Type	Omitted Calls	Runtime Exceptions	Expected Exceptions	Overall
Structural anomalies	17%	4%	9%	10%
HTTP errors	13%	17%	22%	17%

Table 2: Miss-rate comparison of structural anomaly detection and HTTP errors. We omit log file monitoring because of the high false positive rate; in our experiments, some failure was always declared whether or not we injected faults.

observation indicating that playback was about to proceed should have swiftly followed this one, but is instead absent. This pointed the developer to the few lines of code where the stall occurred; he then resolved the bug.

4.1.2 Structural Anomalies

We can also detect failures by searching for anomalies in path structure. We model normal path behavior using a probabilistic context free grammar (PCFG) [34], a structure borrowed from statistical natural language processing. The PCFG models the likelihood of a given path occurring based on the paths seen during training. Magpie [7] also proposes using a PCFG, but has not applied it to anomaly detection.

To generate the PCFG for normal path behaviors, we represent each path as a tree of component function calls. Based on the calls made by each component across all our paths, we generate probabilistic expansion rules, representing the probability that any given component will call a particular set of other components. For example, Figure 4 shows the rules trained from two short paths.

One advantage of the PCFG is that the resultant grammar loosely bounds the acceptable set of paths in our system. Because the grammar is context-free, the learned PCFG actually represents a super-set of the observed paths in the system, providing robustness to false positives. For example, a PCFG model of a user-customizable system, such as a personalizable web

$\mathbf{S} \rightarrow A$	$p = 1.0$	$B \rightarrow C$	$p = 0.5$
$A \rightarrow B$	$p = 0.5$	$B \rightarrow \$$	$p = 0.5$
$A \rightarrow BC$	$p = 0.5$	$C \rightarrow \$$	$p = 1.0$

Figure 4: A sample PCFG, trained on two simple paths: one where A calls B which calls C, and another where A calls both B and C directly. **S** and \$ are the start and end symbols, respectively.

delays impact the user experience. Hiccups on the order of several hundred milliseconds are intolerable along this critical path, so blocking disk writes are performed by another, dedicated thread. This thread also implements a dynamically configurable filter, so that observations within a path are selectively logged based on the structure and latency profile of that path. For example, we may only wish to log a particular observation if another specific observation is present in the same path, or if a pair of observations were made far enough away from each other in time. This way, we can specify greater logging detail for paths of particular interest.

Observations are always recorded internally, so that all path details are available in a core file, but only critical observations (including those required by monitoring logic) are regularly written to disk. Once on disk, ObsLogs are aggregated and stored remotely.

3.3 Analysis Engines and Visualization

We support both single and multi-path analysis, and use dedicated engines to run various statistical tests. Simpler algorithms can be executed in a database, since descriptive statistics such as mean, count, and sum are cheap SQL operations. Some data mining tools are also supported by Oracle [40], so more complicated algorithms such as clustering and classification can be performed, although off-line analysis is a better option at scale. We also use analysis engines written in C++, Java, and Perl, including non-parametric two-sample and analysis of variance (ANOVA) [44] tests.

Visualization is another analysis engine that complements statistical test output to help engineers and operators quickly understand system behavior. We have found Tukey's boxplots[1] useful in summarizing distributions, and survivor plots[2] helpful when focusing on high quantiles and outliers. These plots are generated using GNU Octave [21]. Directed graphs depicting system structure are drawn using Graphviz [6].

4 Failure Management

We now turn to our first path-based analysis application, failure management. Given the inevitability of failures, it is important to improve the mean time to recovery (MTTR) as well as to increase the mean time to failure [10, 19]. The goal of failure management is to minimize failure impact on availability and service quality.

Although proven large-scale system design principles such as replication and layering improve scalability and availability, the resulting componentization impedes failure management, as the execution context is distributed throughout the system. Paths aid in failure management by observing request-centric system behavior, identifying the components involved, linking the symptoms with the distributed state responsible, and providing an effective means of validating system assumptions made while attempting to explain the undesirable behavior.

Paths contribute to failure management at each of the following steps:

1. *Detection*: We first learn of some new, undesired behavior (in QA, development, or production), ideally from an automated analysis engine but sometimes from a human's description.

2. *Isolation*: Analysis engines quickly find representative paths. These tell us which components are involved, and their structure allows us to isolate the problem to a small subset, often just a single component. Such components are immediately cut off from the remainder of the system so they can no longer impact users.

3. *Diagnosis*: A human gets involved at this point, as human response time can no longer impact service quality. The human visualizes several representative paths, and studies the statistically significant deviant component interactions and behaviors. Path-based tools are used to validate or refute hypotheses during the investigation. Path pointers to application-level logs, process images, etc., lead to root cause identification. Paths do not replace traditional local analysis tools, but rather complement them by guiding the diagnosis.

4. *Impact Analysis*: In parallel with diagnosis, the path structure that led to detection is refined so we can use paths to determine the extent of the user impact; i.e., the extent to which other paths exhibit the same problem. This allows us to prioritize the ongoing diagnosis and subsequent repair efforts.

5. *Repair*: We fix the problem.

6. *Feedback*: In parallel with repair, we use the experience as feedback to enhance future detection. If the specific problem could have been caught more quickly with traditional monitoring mechanisms, we now know enough from the paths to implement these. Or perhaps by generalizing our path detection, we could catch a broader class of problems.

We now focus our discussion on three failure management tasks: detection, diagnosis, and impact analysis. We single out some feedback examples as we proceed.

[1]Boxplots illustrate a distribution's center, spread, and asymmetries by using rectangles to show the upper and lower quartiles and the median, and explicitly plotting each outlier [44].

[2]A survivor plot is 1 - CDF, the cumulative distribution function; we plot it with a logarithmic y-axis to add detail for the tail, which is typically the area of interest for latency measurements.

and ends when they hear an aural response. For completeness, we also record path information past these logical endpoints, as shown in Figure 2.

3 Implementation

We describe our path-based macro analysis implementations, organized by the subsystems shown in Figure 1.

3.1 Tracers

Tracers are responsible for tracking a request through the target system and recording any observations made along the way. Our approach is to associate each request with a unique identifier at the system entry point, and to maintain this association throughout. This is similar to Magpie [7] and WebMon, although our tracers record additional information such as resource dependencies and version numbers for failure analysis. We do not record all low-level resources, such as the network packets that Magpie tracks. Although paths may be inferred without them [3, 36], explicit path identifies are essential in linking specific observations with specific failures.

We require that the request identifier be accessible from all areas of the platform that make observations. If threads process requests one at a time, storing IDs in thread-local storage is an attractive option. If protocols with extensible headers are used, such as HTTP or SOAP, we can add the ID as part of a header. Failing that, we can modify the existing protocols, interfaces, class members, etc., so that the request ID follows the control flow.

Alternatively, the entire path state, including the ID and all recorded observations, can be threaded through the system. This simplifies observation aggregation, but there is overhead in moving observations through component boundaries. To optimize performance, we use both techniques: the entire path state where shared memory is available, and just the request ID otherwise.

Although Tracers are platform-specific, they can remain application-generic for platforms that host application components (e.g., J2EE, .NET [35]). This can be done by monitoring requests at application component boundaries, or, more generally, by recording platform-internal notions of content (e.g., URLs, module names, function symbols) along paths.

Pinpoint, ObsLogs, and SuperCal all have platform Tracers to enable monitoring for all applications, without modification. We also provide interfaces for application developers to insert their own data, such as data hashes and references to other data sources, so that in the extreme case, all system logging may be done via paths.

3.1.1 Pinpoint

We modified the Jetty web server to generate a unique ID for each HTTP request. Because Jetty is integrated with JBoss in the same JVM and has a simple threading model, we use Java's ThreadLocal class to store request IDs. When the server invokes a remote Enterprise Java Bean (EJB), our modified Remote Method Invocation protocol passes the ID transparently to the target EJB.

We augmented the Java Server Pages (JSP), Servlets, EJB containers, and JDBC driver to report observations on component and database usage. These observation points report the names of the application component, the methods invoked, and the SQL queries. The total code modification was less than 1K lines, and took about a week of a graduate student's time.

3.1.2 ObsLogs

To minimize performance overhead and optimize for observation insertion, we store the in-process portion of each path in a linked list of fixed-size memory segments. Each observation contains a table index, relative timestamp, and identifier, for a total of 12 bytes. The index identifies the precise location in the system where the observation was made, and the identifier points to other logging sources that provide further contextual information. Recording an observation simply involves copying these 12 bytes into (usually) pre-allocated memory. At current usage levels on modern hardware, there is no statistically significant measurable impact on the CPU usage and end-to-end latencies of the instrumented processes.

The Tellme platform also records how paths interact. Paths may split as parallel subtasks, merge when results are aggregated, or interrupt each other during an abort. All these behaviors are easily tracked since the path state follows the underlying logic: path pointers may be passed to multiple software modules each working on a subtask, code that aggregates results may link one path pointer to another, etc. Using the path identifiers, paths across different processes may be logged in pieces and fully reconstructed by the Aggregator.

3.2 Aggregator and Repository

The Aggregator receives observations from the Tracers, reconstructs them into paths using the request IDs,, and stores them in the Repository. The path data may flow from distributed local storage to a central data repository, and filters may be applied to reduce the data volume, creating a multi-tiered cache for paths.

Pinpoint supports two independent logging modes. The first uses the asynchronous Java Messaging Service to publish observations to (remote) subscribers. In the second mode, observations are written directly to local storage. We use Java's serialization routines to store and retrieve paths from files, and can insert paths into MySQL and Oracle databases over JDBC.

Although Tellme's Tracers frequently process paths with many hundreds of observations, this data rate cannot be sustained with disk writes on the critical path; such

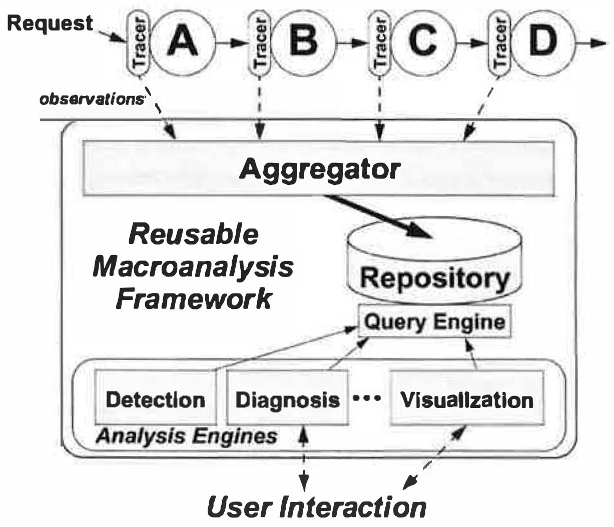


Figure 1: Path-based analysis architecture, illustrating the collection of *observations* from *Tracers* via the *Aggregator* for storage in the *Repository*. Various *Analysis Engines* perform statistical or visual analyses of path data via the *Query Engine*.

developed and run. Legacy applications can take advantage of our path-based tools without modification.

Although Pinpoint is a research prototype, Tellme's real-time, geo-redundant system has serviced many billions of requests since the end of 2001, when ObsLogs (short for Observation Logs) were deployed. eBay services hundreds of millions of requests a day, all of which are monitored using SuperCal, producing hundreds of gigabytes of compressed SuperCal logs.

We will describe our analysis framework before presenting our two main applications: failure management and evolution. In Section 2 we explain our path analysis architecture. We describe relevant implementation details in Section 3. In Section 4, we *detect* failures based on deviations in path structure and interval distributions, show how to accurately and quickly *diagnose* such failures, and gauge their importance via *impact analysis*. We show how to *evolve* the system by deducing structure and regressing changes across system and application versions in Section 5. Finally, we discuss lessons learned in Section 6, and related work in Section 7.

2 Design

Our primary architectural goals are to enable path-based measurement for a variety of systems and to decouple the recording, storage, and analysis functionality, so that these subsystems may scale and evolve independently. Because we are interested in real failures, the framework must be feasible to use on a live system.

In a path-based implementation, a path is a collection of observations, which are local, discrete system measurements at particular points during the system's response to a request. Example observations include

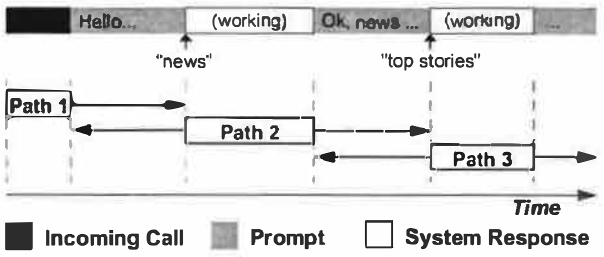


Figure 2: Breaking down a phone call into overlapping paths: each box represents the logical extent of the path, while arrows illustrate the actual range of observations. After connecting, the user says "news", and then later "top stories".

timestamps, component and host names, and version numbers. The observations are recorded by Tracers, shown in Figure 1, and the path data is aggregated, stored, and analyzed statistically or visualized. After defining paths, we describe the main modules of Figure 1 in more detail.

2.1 Defining Paths

A path embodies the control flow, resources, and performance characteristics associated with servicing a request. We use the term "request" in a broad sense for whenever any external entity asks the system to perform some action. The request may result in a response delivered back (e.g., HTTP) or in some action with remote consequences (e.g., UDP packets). Paths may have inter-path dependencies through shared state or resources such as database tables, file systems, or shared memory.

Multiple paths are often grouped together in *sessions*, just as a user's requests collectively contribute to a higher-level goal. Multiple stateless HTTP requests may be tied together with a cookie containing a session ID. For P2P systems, a lookup session may contain several one-way message paths, including queries and result delivery. On Tellme's network, a phone call is a session.

Pinpoint and SuperCal use the natural definition of a path for web services: a web server's response to an HTTP request. The Tellme system paths, shown in Figure 2, need more explanation. VoiceXML end users call a phone number, interact with the system via speech and touch-tone, and ultimately end their call by hanging up or transferring to another number. The user is usually undertaking a high-level application-specific task, such as retrieving driving directions or placing stock trades.

A request-response interaction occurs whenever the user waits for the system to respond. Since the behavior of these response paths directly characterizes a user's experience with the system, and serves as a foundation upon which successful applications can be built, understanding this behavior is extremely important. We thus define paths in Tellme's network as originating when a user provides input and ending when the system responds. For example, a user initiates a path by speaking

Path Framework	Site	Description	Physical Tiers	# of Machines	Live Requests	Apps Hosted
Pinpoint	-	research prototype	2-3	-	-	Java
ObsLogs	Tellme	enterprise voice application network	-	hundreds	millions/day	VoiceXML[55]
SuperCal	eBay	online auction	2-3	thousands	millions/day	C++, Java

Table 1: A comparison of three systems that support path-based analysis.

details, complements and does not replace traditional component-oriented systems approaches. We often use such tools to flesh out issues identified via macro analysis. For example, our failure diagnosis typically can determine the specific requests and component(s) involved in a failure, but resolving the actual cause may require looking at source code or component logs.

In this paper we apply path-based macro analysis to two broad classes of tasks encountered with large, distributed systems: failure management and evolution.

Failure Management consists of the full process of detection, diagnosis, and repair of hardware and software failures. Paths help with three tasks in particular:

Detection: Failures can result in unplanned downtime, and failure detection remains a hard problem, especially at the application level. Tellme Networks, one of the two commercial sites we have analyzed, estimates that roughly 75% of application-level failure recovery time is spent on detection. The difficulty is that overt symptoms caused by abnormal component behavior or bad component interactions may only be visible to the user. However, such problems can impact path structure in many ways, affecting control flow, path latency profiles, and user behavior. Using paths, we can reduce failure detection time and notice developing problems before the consequences become more severe. The key approach is to define "normal" behavior statistically, and then to detect statistically significant deviations.

Diagnosis: Traditionally, once a failure is reported, engineers attempt to reproduce the problem with a simulated workload and verbose logging enabled, and proceed to correlate events in logs on different machines to reconstruct the failure timeline. Our approach differs in that our goal is to isolate problems using solely the recorded path observations, and to subsequently drive the diagnosis process with this path context. This works because correlations across the large number of paths that traverse the system imply which components and requests are involved (and not involved!) in a given failure scenario. This typically requires only a few queries once a failure is detected, and allows us to quickly identify and rank probable root causes. We can also "trap" errors by increasing the amount of detailed logging for isolated components in a live system.

Impact Analysis: After a problem is diagnosed, we would like to understand the impact that it had on users. In this case, we are estimating how many paths have the same profile as the faulty path. Knowing the scale of the problem allows us to prioritize the solution. In the case of failures that affect a Service-Level Agreement (SLA), such as an error in an ad server, impact analysis allows us estimate the damage and determine compensation.

Evolution is challenging for these systems because it is very difficult to replicate the precise timing and behavior of the production system. Thus most upgrades are rolled out in stages to a live system after extensive testing on a smaller "test system" with a simulated load. Systems evolve through both changes to their components and changes in how they interact. Paths help by revealing the actual system structure and dependencies, and tracking how they change. More importantly, our statistical approach allows us to simultaneously detect a wide range of performance and correctness issues across system versions. For each statistically significant change, we can drill down to understand the requests and components affected. This allows for both the validation of *expected* changes as well as the detection and diagnosis of *unexpected* changes.

We present our evaluation of path-based analysis on three implementations, summarized in Table 1.

Pinpoint is an analysis framework for an open-source, 3-tier Java 2 Enterprise Edition (J2EE) [47] application platform, JBoss [30].

ObsLogs are part of a path-based infrastructure at Tellme Networks, an enterprise voice application network.

SuperCal is the logging infrastructure at eBay, an online auction site.

eBay and Tellme are geo-redundant systems. We believe, but do not show, that paths apply equally well to wide-area distributed systems, including peer-to-peer networks and sensor networks. The primary limitation of our approach is the need to aggregate the logs for analysis (described in the next two sections), as we do not present a wide-area distributed query system.

For our purposes, JBoss, eBay, and Tellme's network can be considered cluster-based application servers that provide a platform upon which applications are

Path-Based Failure and Evolution Management

Mike Y. Chen, Anthony Accardi, Emre Kıcıman, Jim Lloyd, Dave Patterson, Armando Fox, Eric Brewer
UC Berkeley, Tellme Networks, Stanford University, eBay Inc.
{mikechen, patterson, brewer}@cs.berkeley.edu, anthony@tellme.com, {emrek, fox}@cs.stanford.edu, jlloyd@ebay.com

Abstract

We present a new approach to managing failures and evolution in large, complex distributed systems using runtime paths. We use the paths that requests follow as they move through the system as our core abstraction, and our "macro" approach focuses on component interactions rather than the details of the components themselves. Paths record component performance and interactions, are user- and request-centric, and occur in sufficient volume to enable statistical analysis, all in a way that is easily reusable across applications. Automated statistical analysis of multiple paths allows for the detection and diagnosis of complex failures and the assessment of evolution issues. In particular, our approach enables significantly stronger capabilities in failure detection, failure diagnosis, impact analysis, and understanding system evolution. We explore these capabilities with three real implementations, two of which service millions of requests per day. Our contributions include the approach; the maintainable, extensible, and reusable architecture; the various statistical analysis engines; and the discussion of our experience with a high-volume production service over several years.

1 Introduction

The rise of large, highly available, networked systems [10, 26] reinforces a trend towards complex, heterogeneous architectures composed of distributed, replicated components. Such systems may be built from thousands of machines, each running a diverse set of software components that exhibit complicated interactions [18, 23]. This trend undermines basic system management tasks, from detecting and diagnosing failures to understanding current and future system behavior. Although there are many tools for dealing with individual components, such tools are less effective in understanding aggregate system behavior, and worse, lose sight of the impact of specific components on the user experience.

Existing monitoring and debugging techniques use tools such as code-level debuggers, program slicing [53], code-level and process profiling [22, 31, 42], and application-level logs. Although these techniques provide valuable information about individual components, this localized knowledge fails to capture the component interactions that characterize the overall system behavior and determine the user experience. Although some tools, such as distributed debuggers, cover multiple components, they focus on a homogeneous subset of the system and usually consider one node at a time. Some such tools require additional component knowledge, which may be difficult to obtain for "black box" components.

Our goal is to design tools that help us understand large distributed systems to improve their availability, reliability, and manageability. We trace paths from user requests, through distributed black-box components, until service completes. Examples include the request/response interaction in Internet systems and one-way flows in messaging systems. We apply statistical techniques to the data collected along these paths to infer system behavior. We draw on two main principles:

Path-Based Measurement: We model the target system as a collection of paths through abstract, black-box, heterogeneous components. Local observations are made along these paths, which are later accessed via query and visualization mechanisms.

Statistical Behavior Analysis: Large volumes of system requests are amenable to statistical analysis. We use classical techniques to automatically identify statistically significant deviations from normal behavior for both performance and correctness, and for both live system operation and off-line analysis.

The path abstraction has been used in many other areas, typically as a form of control flow. Paths are used in Scout for a single-node OS [38], are the foundation for integrated-layer-processing [1], and play a key role in many compiler optimization techniques [5]. Several recent systems have also used paths to profile distributed systems. Magpie [7] and WebMon [49] both trace web requests across multi-tiered web systems for performance tuning and diagnosis. Aguilera *et al.* present both statistical and message-by-message algorithms to infer causal paths and thereby debug performance problems in distributed systems of black boxes [3].

Our use of paths is novel in that we focus on correctness rather than performance. We use paths to detect and diagnose failures, and to understand the evolution of a system. Although we have also applied paths to profile the performance of two large production services, this work is less novel and we omit it for space.

We stress that our "macro" approach [12], where we focus on component-level abstractions over component services and encourage others working on self-managing systems to explore similar recovery-friendly designs.

11. REFERENCES

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proc. Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Monterey, CA, 2002.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, pages 33–38, Elmau/Oberbayern, Germany, May 2001.

[3] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[4] E. Brewer. Running Inktomi. Personal Communication, 2001.

[5] A. B. Brown and D. A. Patterson. To err is human. In *Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY)*, Göteborg, Sweden, July 2001. IEEE Computer Society.

[6] G. Candea and A. Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[7] W. Clinger and L. Hansen. Generational garbage collection and the radioactive decay model. *Proc. SIGPLAN 97 Conference on Programming Language Design and Implementation*, May 1997.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–12, 1989.

[9] FAUMachine. `http://www.FAUmachine.org/`.

[10] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[11] R. Folwer, S. E. Alan Cox, and W. Zwaenepoel. Using performance reflection in systems software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[12] A. Fox and E. A. Brewer. ACID confronts its discontents: Harvest, yield, and scalable tolerant systems. In *Seventh Workshop on Hot Topics In Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[13] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Enterprise storage systems on a shoestring. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[14] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pages 282–292, Paderborn, Germany, Nov 1998.

[15] D. K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1979.

[16] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, 1989.

[17] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[18] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *Proc. International Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003.

[19] A. C. Huang and A. Fox. Decoupling state stores for ease of management. Submitted to FAST, 2004.

[20] D. Jacobs. Personal Communication, 2003.

[21] D. Jacobs. Distributed computing with BEA WebLogic server. In *Proceedings of the Conference on Innovative Data Systems Research*, Asilomar, CA, Jan. 2003.

[22] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM Conference*, Stanford, CA, Aug. 1988.

[23] E. Keogh, S. Lonardi, and W. Chiu. Finding surprising patterns in a time series database in linear time and space. In *In proc. of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 550–556, Edmonton, Alberta, Canada, Jul 2002.

[24] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[25] B. Ling and A. Fox. The case for a session state storage layer. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[26] B. C. Ling and A. Fox. A self-tuning, self-protecting, self-healing session state management layer. In *5th Annual Workshop On Active Middleware Services*, Seattle, WA, 2003.

[27] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, CA, Oct 1991.

[28] Network Appliance. `http://www.netapp.com/`.

[29] T. Networks.

[30] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[31] A. Pal. Yahoo! User Preferences Database. Personal Communication, 2003.

[32] J. Ping, Z. Ge, J. Kurose, and D. Towsley. A comparison of hard-state and soft-state protocols. In *Proceedings of the ACM SIGCOMM Conference*, pages 251–262, Karlsruhe, Germany, Aug 2003.

[33] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, Sept. 1999.

[34] Resin. `http://www.caucho.com/`.

[35] T. Roscoe and S. Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, June 2003.

[36] Silk Performer. `http://www.segue.com/`.

[37] U. Singh. E.piphany. Personal Communication, 2003.

[38] A. Vermeulen. Personal communication, June 2003.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Lake Louise - Banff, Canada, 2001.

does not require coordination with other bricks.

As a result, the monitoring system that detects failures is allowed to make mistakes. In contrast, in other systems, false positives usually reduce performance, lower throughput, or cause incorrect behavior. Since false positives are not a problem in SSM, generic methods such as statistical anomaly based failure detection can be made quite aggressive, to avoid missing real faults.

8. RELATED WORK

Palimpsest [35] describes a scheme for temporal storage for planetary-scale services. Palimpsest requires a user to erasure-code the relevant data, and write it to N replica sites, which may all be under separate administrative control. Like SSM, all metadata for the write is stored on the client. However, Palimpsest is intended for the wide area; storage sites may be under different administrative domains. Palimpsest gives no guarantees to its users in terms of storage lifetime; SSM gives probabilistic guarantees.

Several projects have focused on the design and management of persistent state stores [17, 28, 1, 13, 19]. FAB [13] shares many of the same motivations as SSM, including ease of management and recovery; however, FAB is intended at a very different level in the storage hierarchy. FAB is a logical disk system for persistent storage that is intended to replace enterprise-class disk arrays, while in SSM we focus on temporal storage of session state. In addition, in SSM, all metadata is stored at the client, while FAB employs a majority-voting algorithm. Similarly, DStore [19] shares many of the motivations as SSM, but it focuses on unbounded-persistence storage for non-transactional, single-key-index state.

Petal [24] attempts to make data highly available by placing data on a node, and placing a backup copy on either the predecessor or the successor. Upon failure, the load is divided by the predecessor and successor, whereas in SSM the load redistribution is even across all nodes. In Petal, the loss of any two adjacent nodes implies data loss, while in SSM, the number of replicas is configurable.

SSM's algorithm is different from that of quorums [15]. In quorum systems, writes must be propagated to W of the nodes in a replica group, and reads must be successful on R of the nodes, where $R + W > N$, the total number of nodes in a replica group. A faulty node will often cause reads to be slow, writes to be slow, or possibly both. Our solution obviates the need for such a system, since the cookie contains the references to up-to-date copies of the data.

DDS [17] is similar to SSM in that it focuses on state accessible by single-key lookups. A detailed discussion of the differences between SSM and DDS can be found in previous work [25, 26]. We share many of the same motivations as Berkeley DB [30], which stressed the importance of fast-restart and treating failure as a normal operating condition, and recognized that the full generality of databases is sometimes unneeded.

The windowing mechanism used by the stubs is motivated by the TCP algorithm for congestion control [22]. The need to include explicit support for admission control and overload management at service design time was demonstrated in SEDA [40]; we appeal to this argument in our use of windowing to discover the system's steady-state capacity and our use of "backpressure" to do admission control to prevent driving the system over the saturation cliff.

Zwaenepoel et al [11] are also looking at using generic, low-level statistical metrics to infer high-level application behaviors. In their case, they are looking at CPU counters such as number of instructions retired and number of cache misses to make inferences about the macro-level behavior of the running application.

9. FUTURE WORK

SSM currently does not tolerate catastrophic site failures, but can be extended to do so. When selecting bricks for writes, SSM can be extended to select W_{local} bricks from the local network, and W_{remote} bricks from a remote site. SSM can return from writes when WQ_{local} bricks have replied, and 1 remote brick has replied.

Intelligently shedding load is an area of active research. One policy is to allow only users that are already actively using the system to continue using the system, and to turn new sessions away; this can be done by only allowing writes by users that have valid cookies when the system is overloaded. Alternatively, users can be binned into different classes in some external fashion, and under overload, SSM can be configured to service only selected classes.

We are exploring the use of rolling reboots as a method of proactively avoiding failures.

Currently, Pinpoint monitors statisics that empirically correlate with injected failures; however, we have no proof that they are the most relevant ones. We intend to apply statistical learning theory to automatically determine which measurable features best correlate with failures.

10. CONCLUSIONS

A "new wave" of systems research is focusing on the dual synergistic areas of reduced total-cost-of-ownership and managed/hosted online services. Many groups have proposed visions for self-managing, self-adapting, or self-healing systems; we have presented an architecture and implemented prototype that realizes some of those behaviors in a state-storage subsystem for online services. We have also attempted to illuminate one approach to self-management in the context of this work: make recovery so fast and cheap that false positives during failure detection become less important, thereby allowing the use of powerful, self-adapting, application-generic failure detection techniques such as statistical-anomaly analysis. We hope that SSM will both prove useful as a building block for future online

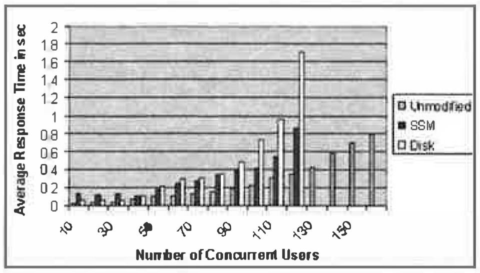


Figure 14: **Latency vs. load for 10 to 160 users. The original application can handle a capacity of 160 simultaneous users. The modified application using disk or using SSM can each handle 120 users.**

store. The modified application reaches capacity at 120 concurrent users, with an average response time of 1.72 seconds.

Lastly, we integrate SSM with the application. Three other machines are configured as bricks. A switch sits between the three bricks and the application server. The integrated application reaches capacity at 120 simultaneous users, with an average response time of 0.863 seconds.

Figure 14 summarizes the results. Compared to storing session state in-memory-only, using our prototype of SSM imposes a 25 percent throughput penalty on the overall application: the maximum number of simultaneous users is reduced from 160 to 120, although the per-user response times are roughly equal in the two cases, so users perceive no latency penalty.

Compared to using a filesystem, SSM supports just as many concurrent users, but delivers better response time: with 120 active users, the application using disk runs more than twice as slowly as the application using SSM.

In summary, integrating SSM with the application imposes a 25 percent throughput overhead compared to in-memory-only, but preserves throughput and delivers better response time than the disk solution. Neither the in-memory nor the filesystem solution provide SSM's high availability, self-recovery and self-healing.

7. DISCUSSION

SSM bricks can be built from simple commodity hardware. From a few months experience working with SSM, bricks perform very predictably, which in turn allows detection of anomalies to be extremely simple and accurate. In this section we try to distill what properties of SSM's design and algorithms give rise to these properties.

7.1 Eliminate Coupling

In SSM, we have attempted to eliminate all coupling between nodes. Bricks are independent of other bricks, which are independent of stubs. Stubs are independent of all other stubs; each stub is regulated by an AIMD sending window which prevents it from saturating the system.

In traditional storage systems, a requestor is coupled to a requestee, and the requestor's performance, correctness, and availability are all dependent on the requestee. SSM instead uses single-phase, non-locking operations, allowing writes to proceed at the speed of the fastest WQ bricks instead of being coupled to lagging or failing bricks. Among other things, this makes lengthy garbage collection times unimportant, since a brick performing GC can temporarily fall behind without dragging down the others.

Elimination of coupling comes at a cost: redundancy is harnessed for performance. Redundant components with reduced coupling gives rise to predictable performance. Coupling elimination has been used in [19, 12, 4].

Related to coupling is the use of both randomness to avoid deterministic worst cases and overprovisioning to allow for failover. Both techniques are used in large-system load balancing [3], but SSM does this at a finer grain, in the selection of the write set for *each request*.

7.2 Make Parts Interchangeable

For a write in SSM, any given brick is as good as any other in terms of correctness. For a read, a candidate set of size R is provided, and any brick in the candidate set can function in the place of any other brick. The introduction of a new brick does not adversely disrupt the existing order; it only serves to increase availability, performance, and throughput.

In many systems, certain nodes are required to fulfill certain fixed functions. This inflexibility often causes performance, throughput or availability issues, as evidenced in DDS [17], which uses the buddy system.

In SSM, bricks are all equivalent. Because complete copies of data are written to multiple bricks, bricks can operate independently, and do not require any sort of coordination. Furthermore, as long as one brick from the write quota of size WQ remains, the data is available, unlike in erasure coding systems such as Palimpsest [35], where a certain number of chunks is required to reconstruct data.

7.3 It's OK to Say No

SSM uses both adaptive admission control and early rejection as forms of backpressure. AIMD is used to regulate the maximum number of requests a stub can send to a particular brick; each brick can reject or ignore timed-out requests; the application of TCP, a well-studied and stable networking protocol, allows SSM components to reach capacity without collapse [40]. The goal of these mechanisms is to avoid having SSM attempt to give a functionality guarantee ("I will do it") at the expense of predictable performance; the result is that each request requires a predictable amount of work.

7.4 It's OK to Make Mistakes

The result of the application of redundancy and the interchangeability of components is that recovery is fast, simple, and unintrusive: a brick is recovered by rebooting it without worrying about preserving its pre-crash state, and recovery

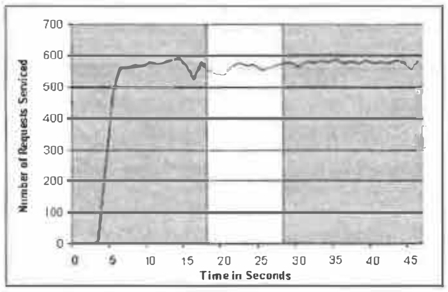


Figure 11: **Fault Injection: Memory Bitflip in hashtable**

and we increase t to 700ms and decrease the size of the state written to 3KB to adjust to the order of magnitude slowdown.

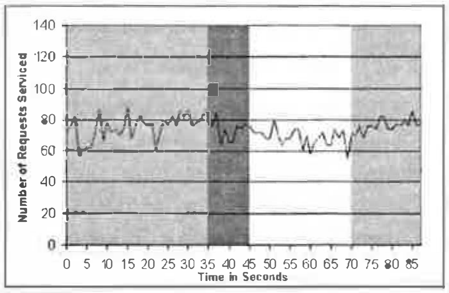


Figure 12: **Fault Injection: Dropping 70 percent of outgoing packets. Fault injected at time 35, brick killed at time 45, brick restarted at time 70.**

The fault is injected at time 35; however, the brick continues to run with the injected fault for 10 seconds, as shown in the darkened portion of figure 12. At time 45, Pinpoint detects and kills the faulty brick. The fault is cleared to allow network traffic to resume as normal, and the brick is restarted. Restart takes significantly longer using the FAUMachine, and the brick completes its restart at time 70. During the entire experiment, all requests complete correctly in the specified timeout and data is available at all times. Throughput is affected slightly, as expected, as only five bricks are functioning during times 45-70; recall that running bricks on FAUMachine causes an order of magnitude slowdown.

5.8 CPU/Memory Performance Fault

SSM is able to tolerate performance faults, and Pinpoint is able to detect performance faults and reboot bricks accordingly. In the following benchmark with 6 bricks and 3 load-generating machines, we inject performance failures in a single brick by causing the brick to sleep for 1ms before handling each message. This per-message performance failure simulates software aging. In figure 13, we inject the fault every 60 seconds. Each time the fault is injected, Pinpoint detects the fault within 5-10 seconds, and reboots the brick. All requests are serviced properly.

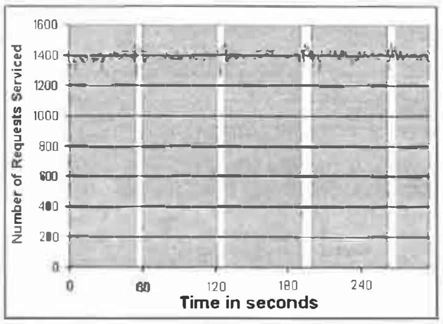


Figure 13: **Performance Fault: Brick adds 1ms sleep before each request; faults injected every 60 seconds, Pinpoint detects failure within 5-10 seconds, and brick is restarted.**

6. END TO END APPLICATION BENCHMARKS

In this section, we integrate SSM with a production, enterprise-scale application. We also modify the application to use disk to store session state, as a baseline comparison. We compare the integrated solution with the unmodified application, as well as the application modified to use disk to store session state.

The application we use is a simplified version of Tellme's [29] Email-By-Phone application; via the phone, users are able to retrieve their email and listen to the headers of various folders by interacting with voice-recognition and voice-synthesis systems integrated with the email server. The application logic itself is written in Java and run on Resin [34], an XML application server on top of a dual processor Pentium III 700 MHz machine with 1G RAM, running Solaris. We use 3 bricks for the benchmark. All machines are connected via switched ethernet, and held in a commercial hosting center.

Session state in this application consists of the index of the message the user is currently accessing, the name of the folder that is currently being accessed, and other workflow information about the user's folders. In the original application, the session state is stored in memory only; a crash of the application server implies a loss of all user sessions, and a visible application failure to all active users.

We use Silk Performer [36] to generate load to the application. The load generator simulates users that start the application, listen to an email for three seconds, and progress to the next email, listening to a total of 20 emails. The test is intended to establish a baseline for response time and throughput for the unmodified application. We vary the load from ten users to 160 users; at 170 users, the application server begins throwing "Server too busy" errors. Unmodified, the application server can handle 160 simultaneous users, average a response time of 0.793 seconds.

We modify the application to write session state to disk, to establish comparison values for an external persistent state

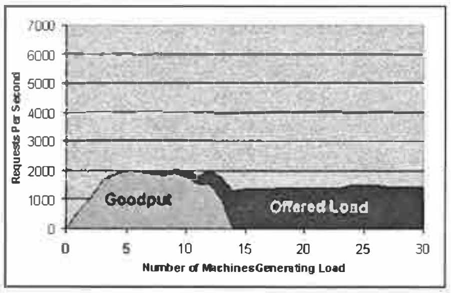


Figure 8: **Steady state graph of load vs. goodput. SSM running without self-protecting features. Goodput peaks at around 1900 requests per second. Half of system goodput is reached at 13 load generating machines, and system goodput drops to 0 at 14 machines.**

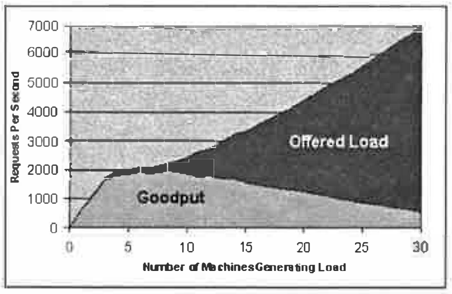


Figure 9: **Steady state graph of load vs. goodput. SSM running with self-protecting features. Goodput peaks at around 1900 requests per second. Half of system goodput is reached at 24 load generating machines, and system goodput trends to 0 at 37 machines.**

without affecting correctness and availablility, and to a degree, performance and throughput, coupled with a generic fault-detecting mechanism such as Pinpoint, gives rise to a self-healing system.

As discussed earlier, in SSM, a single brick can be restarted without affecting correctness, performance, availability, or throughput; the cost of acting on a false positive on SSM is very low, so long as the system does not make false positive errors with too high a frequency. For transient faults, Pinpoint can detect anomalies in brick performance, and restart bricks accordingly.

The following microbenchmarks demonstrate SSM's ability to recover and heal from transient faults. We attempt to inject realistic faults for each of SSM's hardware components—processor, memory, and network interface. We assume that for CPU faults, the brick will hang or reboot, as is typical for most such faults [18].

To model transient memory errors, we inject bitflip errors into various places in the brick's address space. To model a faulty network interface, we use FAUMachine [9], a Linux-based VM that allows for fault-injection at the network level, to drop a specified percentage of packets. We also model performance faults, where one brick runs more slowly than the others. In all of the following experiments, we use six bricks; Pinpoint actively monitors all of the bricks.

5.5 Memory Fault in Stack Pointer

SSM heals itself in the presence of a memory fault; performance and throughput is unaffected, and SSM recovers from the fault.

Using ptrace(), we monitor a child process and change its memory contents. In this benchmark, $W = 3, WQ = 2, R = 2$, data size is 8KB, and we increase t to 100ms to account for the slowdown of bricks using ptrace. Figure 10 shows the results of injecting a bitflip in the area of physical memory where the stack pointer is held. The fault is injected at time 14; the brick crashes immediately. The lightened section of figure 10 (time 14-23) is the time during which only five bricks are running. At time 23, Pinpoint detects that the brick has stopped sending heartbeats and should be restarted, and restarts the brick; the system tolerates the fault and successfully recovers from it.

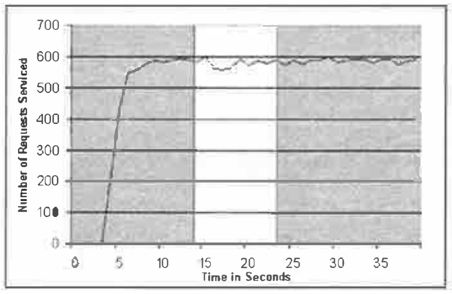


Figure 10: **Fault Injection: Memory Bitflip in Stack Pointer.**

5.6 Memory Fault in Data Value/Checksum

SSM heals itself in the presence of a memory fault in its internal data structures. Figure 11 shows the injection of a bitflip in the object of a session state object that has just been written and is about to be read. The fault is injected at time 18. The brick is configured to exit upon the detection of a checksum error, and does so immediately at time 18. The lightened section of figure 11 (time 18-29) is the time during which only five bricks are running. At time 29, Pinpoint detects that the brick has stopped sending heartbeats and should be restarted, and restarts the brick; the system tolerates the fault and successfully recovers from it.

5.7 Network Performance Fault

SSM can tolerate and recover from transient network faults. We use FAUMachine to inject a fault at the brick's network interface. In particular, we cause the brick's network interface to drop 70 percent of all outgoing packets. Figure 12 shows this experiment running on FAUmachine. Note that FAUmachine overhead causes the system to perform an order of magnitude slower; we run all six bricks on FAUMachine. In this benchmark, $W = 3, WQ = 2, R = 2$, We generate an even higher load than what caused goodput to fall to zero in the basic system, using 240 threads, corresponding to roughly 4000 requests per second. As seen in figure 7, SSM discovers the maximum goodput and the system continues to operate at that level. Note that this means that the system is rejecting the excess requests, since the bricks are already at capacity, and the excess load is simply being rejected; the percentage of rejected requests is discussed in the next section. We sketch a simple and reasonable shedding policy in future work.

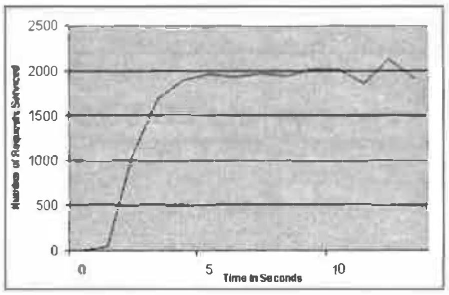


Figure 7: **SSM running with 3 Bricks, with AIMD and admission control. SSM discovers maximum goodput of around 2000.**

5.3 Self-Protecting

SSM protects its components from collapsing under overload. The use of AIMD and admission control allow SSM to protect itself. In particular, the maximum allowable pending non-acked requests that a stub can generate for a particular brick is regulated by the sending window size, which is additively increased on success and multiplicatively decreased on failure. This prevents the stubs from generating load that results in brick overload; each stub exerts backpressure [40] on its caller when the system is overloaded. In addition, bricks actively discard requests that have already timedout, in order to service only requests that have the potential of doing useful work.

SSM protects itself under overload. Part of the self-protecting aspect of SSM is demonstrated in the previous benchmark; SSM's goodput does not drop to zero under heavier load. In this benchmark, W is set to 3, WQ is set to 2, timeout is set to 60 ms, R is set to 1, and the size of state written is 8K.

SSM's use of the self-protecting features allows SSM to maintain a reasonable level of goodput under excess load. Figure 8 shows the steady state graph of load vs. goodput in the basic system without the self-protecting features. Figure 9 shows the steady state graph of load vs. goodput in SSM with the self-protecting features enabled. The x-axis on both graphs represents the number of load-generating machines; each machine runs 12 threads. The y-axis represents the number of requests. We start with the load generator running on a single machine, and monitor the goodput of SSM after it has reached steady state. Steady state is usually reached in the first few seconds, but we run each configuration for 2 minutes to verify that steady state behavior remains the same. We then repeat the experiment by increasing the number of machines used for load generation.

Comparison of the two graphs shows:

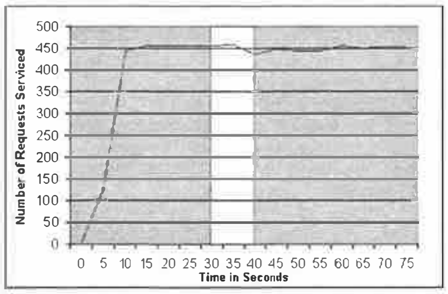
- The self-protecting features protect the system from overload and falling off the cliff and allows the system to continue to do useful work.
- Extends useful life of the system under overload. Without the self-protecting features, we see that maximum goodput is around 1900 requests per second, while goodput drops to half of that at a load of 13 machines, and falls to zero at 14 machines. With the self-protecting features, maximum goodput remains the same, while goodput drops to half of the maximum at 24 machines, and goodput trends to zero at 37 machines, because SSM begins spending the bulk of its processing time trying to protect itself and turning away requests and is unable to service any requests successfully. With self-protecting features turned on, the system continues to produce half of goodput at 24 machines vs. 13 machines, protecting the system from almost double the load.

Note that in Figure 8 where goodput has dropped to zero, as we increase the number of machines generating load that the offer load increases only slightly, staying around 1500 failed requests per second. This is because each request must wait the full timeout value before returning to the user; the requests that are generated will arrive at the bricks, but will not be serviced in time. However, in Figure 9, the number of failed requests increases dramatically as we increase the number of machines. Recall that the load generator models hyperactive users that continually send read and write requests; each user is modeled by a thread. When one request returns, either successfully or unsuccessfully, the thread immediately generates another request. Because SSM is self-protecting, the stubs say "no" to requests right away; under overload, requests are rejected immediately. The nature of the load generator then causes another request to be generated, which is likely to be rejected as well. Hence the load generator continues to generate requests at a much higher rate than in Figure 8 because unfulfillable requests are immediately rejected.

5.4 Self-Healing

The ability of a system to heal itself without requiring administrator assistance greatly simplifies management. However, accurate detection of faults is difficult. Usually, acting on an incorrect diagnosis such as a false positive results in degraded system performance, availability, correctness, or throughput. In SSM, the ability to reboot any component

We run four bricks in the experiment, each on a different ph sica machine in the c uster We use a sin e machine as the oad enerator, ith ten orker threads eneratin re uests at a rate of approximate re uests per second



i ure **SSM running with 4 Bricks. One brick is killed manually at time 30, and restarted at time 40. Throughput and availability are unaffected. Although not displayed in the graph, all requests are all fulfilled correctly, within the specified timeout.**

We induce a fault at time 30 by killing a brick by hand. As can be shown from the graph, throughput remains unaffected. Furthermore, all requests complete successfully; the load generator showed no failures. This microbenchmark is intended to demonstrate the recovery-friendly aspect of SSM. In a non-overloaded system, the failure and recovery of a brick has no negative effect on correctness, system throughput, availability, or performance. All generated requests completed within the specified timeout, and all requests returned successfully.

5.2 Self-Tuning

The use of AIMD allows the stubs to adaptively discover the capacity of the system, without requiring an administrator to configure a system and a workload, and then run experiments to determine whether the system services the workload in an acceptable fashion. In the manual process, if the workload increases drastically, the configuration may need to be changed.

In SSM, the allowable amount of time for session state retrieval and storage is specified in a configurable timeout value. The system tunes itself using the AIMD mechanism, maximizing the number of requests that can be serviced within that time bound. SSM automatically adapts to higher load gracefully. Under overload, requests are rejected instead of allowing latency to increase beyond a reasonable threshold. If SSM is is deployed in an environment with a pool of free machines, Pinpoint can monitor the number of requests that are rejected, and start up new bricks to accomodate the increase in workload.

SSM discovers the maximum throughput of the system correctly. Recall that in SSM, read and write requests are expected to complete within a timeout. We define *goodput* as the number of requests that complete within the specified timeout. Offered load is goodput plus all requests that fail. Requests that complete after that timeout are not counted toward goodput. In this benchmark, W is set to 3, WQ is set to 2, timeout is set to 60 ms, R is set to 1, and the size of state written is 8K. We use 3 bricks.

First, we discover the maximum goodput of the basic system with no admission control or AIMD. We do so by varying the number of sending threads to see where goodput plateaus. We run separate experiments; first we generate a load with 120 threads corresponding to roughly 1900 requests per second, and then with 150 threads, corresponding to roughly 2100 requests per second. Figure 5 shows that goodput plateaus around 1900-2000 requests per second.

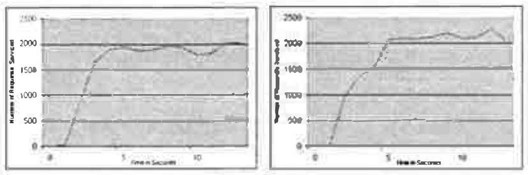


Figure 5: **SSM running with 3 Bricks, no AIMD or admission control. The graph on the left shows a load of 120 threads sending read and write requests of session state. The graph on the right shows a load of 150 threads. sending threads. System throughput peaks at around 1900-2000 requests per second.**

We continue increasing the load until goodput drops to zero. Goodput eventually drops to zero because the rate of incoming requests is higher than the rate at which the bricks can process, and eventually, the brick spends all of its time fulfilling timed-out requests instead of doing useful work. As can be seen in the lightened portion of Figure 6, the bricks collapse under the load of 220 threads, or about 3400 requests a second; requests arrive at a rate faster than can be serviced, and hence the system goodput falls to zero at time 11.

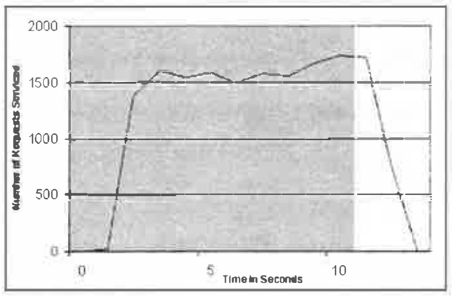


Figure 6: **SSM running with 3 Bricks, no AIMD or admission control. The bricks collapse at time 11 under the load of 220 threads generating requests.**

After manually verifying the maximum goodput of the system, we turn on the self-protecting features, namely by allowing the stub to use the AIMD sending window size and by forcing bricks to service only requests that have not timed out, and run the experiment again.

metric is robust to outliers even in small populations, and lets us identify deviant activity statistics with a low-false positive rate.

State statistics represent the size of some state, such as the size of the message inbox. In SSM, these state statistics often vary in periodic patterns, e.g., in normal behavior, the MemoryUsed statistic grows until the garbage collector is triggered to free memory, and the pattern repeats. Unfortunately, we do not know *a priori* the period of this pattern – in fact, we cannot even assume a regular period.

To discover the patterns in the behavior of state statistics, we use the Tarzan algorithm for analyzing time series [23]. For each state statistic of a brick, we keep an N-length history or time-series of the state. We discretize this time-series into a binary string. To discover anomalies, Tarzan counts the relative frequencies of all substrings shorter than k within these binary strings. If a brick's discretized time-series has a surprisingly high or low frequency of some substring as compared to the other brick's time series, we mark the brick as potentially faulty. This algorithm can be implemented in linear time and linear space, though we have found we get sufficient performance from our simpler non-linear implementation.

Once a brick has been identified as potentially faulty through three or more activity and state statistics, we conclude that the brick has indeed failed in some way; Pinpoint restarts the node. In the current implementation, a script is executed to restart the appropriate brick, though a more robust implementation might make use of hardware-based leases that forcibly reboot the machine when they expire [10].

Because restarting a brick will only cure transient failures, if Pinpoint detects that a brick has been restarted more than a threshold number of times in a given period, which is usually indicative of a persistent fault, it can take the brick off-line and notify an administrator.

5. EXPERIMENTAL RESULTS

In this section, we highlight the key features from the design of SSM. We present benchmarks illustrating each of the recovery-friendly, self-tuning, and self-protecting features. We also present numerous benchmarks demonstrating the self-healing nature of SSM when integrated with Pinpoint.

Each benchmark is conducted on the UC Berkeley Millennium Cluster. Our load generator models hyperactive users who continually make requests to read and write session state; each hyperactive user is modeled using a thread which does a sequence of alternating write and read requests, which is representative of the workload for session state, as described earlier in Section 2. As soon as a request returns, the thread immediately makes a subsequent request. In the following benchmarks, we vary the number of sending threads as well as the number of bricks. All bricks are run on separate, dedicated machines.

5.1 Recovery-Friendly

In a sufficiently-provisioned, non-overloaded system, the failure and recovery of a single brick does not affect:

- Correctness. As described above, the failure of a single brick does not result in data loss. In particular, SSM can tolerate $WQ - 1$ simultaneous brick failures before losing data.

  A restart of the brick does not impact correctness of the system.
- Performance. So long as W is chosen to be greater than WQ and R is chosen to be greater than 1, any given request from a stub is not dependent on a particular brick. SSM harnesses redundancy to remove coupling of individual requests to particular bricks.

  A restart of the brick does not impact performance; there is no special case recovery code that must be run anywhere in the system.
- Throughput. A failure of any individual brick does not degrade system throughput in a non-overloaded system. Upon first inspection, it would appear that all systems should have this property. However, systems that employ a buddy system or a chained clustering system [17, 24] fail to balance the resulting load evenly. Consider a system of four nodes A, B, C, and D, where A and B are buddies, and C and D are buddies. If each node services load at 60 percent of its capacity and subsequently, node D fails, then its buddy node C must attempt to service 120 percent of the load, which is not possible. Hence the overall system throughput is reduced, even though the remaining three nodes are capable of servicing an extra 20 percent each.

  Because the resulting load is distributed evenly between the remaining bricks, SSM can continue to handle the same level of throughput so long as the aggregate throughput from the workload is lower than the aggregate throughput of the remaining machines.

  The introduction of a new brick or a revived brick never decreases throughput; it can only increase throughput, as new bricks add new capacity to the system. A newly restarted brick, like every other brick, has no dependencies on any other node.
- Availability. In SSM, all data is available for reading and writing during both brick failure and brick recovery. In other systems such as unreplicated file systems, data is unavailable for reading or writing during failure. In DDS [17] and in Harp [27], data is available for reading and writing after a node failure, but data is not available for writing during recovery because data is locked and is copied to its buddy en masse.

SSM is recovery-friendly. In this benchmark, W is set to 3, WQ is set to 2, *timeout* is set to 60 ms, R is set to 2, and the size of state written is 8K.

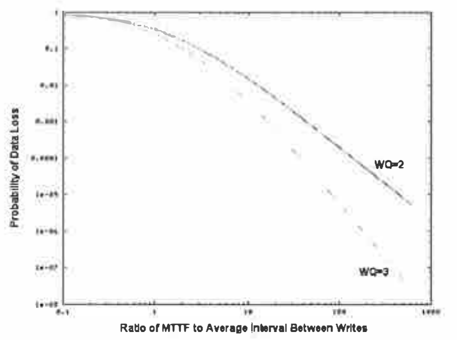


Figure 2: **Probability of data loss with WQ=2 and 3. The x-axis is the ratio of MTTF to the session expiration time. The y-axis is the probability that all WQ copies are lost before the subsequent write.**

that session state would not be expected to survive [20]. We describe a natural extension to SSM to survive site failures in section 8.

Let brick failure be modeled by a Poisson process with rate μ (i.e., the brick's MTTF is $1/\mu$), and let writes for a particular user's data be modeled by a Poisson process with rate λ. (In other words, in practice $1/\lambda$ is the session expiration time, usually on the order of minutes or tens of minutes.) Then $\rho = \lambda/\mu$ is intuitively the ratio of the write rate to the failure rate, or equivalently, the ratio of the MTTF of a brick vs. the write interarrival time.

A session state object is lost if all WQ copies of it are lost. Since every successful write re-creates WQ copies of the data, the object is *not* lost if at most $WQ - 1$ failures occur between successive writes of the object. Equations 1 and 2 show this probability for $WQ = 3$ and $WQ = 2$ respectively; figure 2 shows the probabilities graphically.

$$P^{WQ=3}_{noloss} = \frac{\rho(\rho^2 + 6\rho + 11)}{(\rho+1)(\rho+2)(\rho+3)} \quad (1)$$

$$P^{WQ=2}_{noloss} = \frac{\rho(\rho+3)}{(\rho+1)(\rho+2)} \quad (2)$$

Table 2 summarizes the implication of the equations in terms of "number of nines" of availability. For example, to achieve "three nines" of availability, or probability 0.999 that data will not be lost, a system with $WQ = 2$ must be able to keep an individual brick from crashing for an interval that is 43.3 times as long as the average time between writes. Adding redundancy ($WQ = 3$) reduces this, requiring an MTTF that is only 16.2 times the average time between writes. For example, if the average time between writes is 5 minutes and $WQ = 3$, three nines can be achieved as long as brick MTTF is at least 81 minutes.

Another way to look at it is to fix the ratio of MTTF to the write interval. Figure 3 sets this ratio to 10 (intuitively, this means roughly that writes occur ten times as often as failures) and illustrates the effect of adding redundancy (modifying WQ) on data loss.

	$WQ = 2$	$WQ = 3$
1 Nine	3	2
2 Nines	12.7	6.5
3 Nines	43.3	16.2
4 Nines	140	37.2
5 Nines	446.8	82.4

Table 2: For WQ=2 and 3, the necessary ratio of MTTF to average interval between writes in order for probability of a subsequent write to achieve a certain number of nines

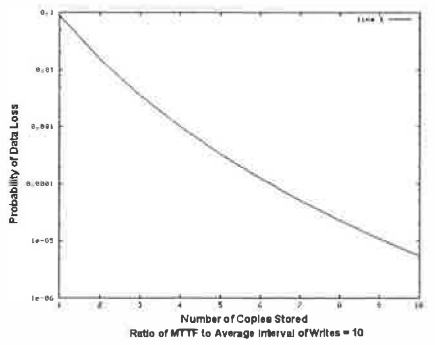


Figure 3: **We fix the ratio of MTTF to the average interval between writes to 10. The x-axis represents the number of copies written. The y-axis represents the probability that all copies are lost.**

4. PINPOINT + SSM = SELF-HEALING

Pinpoint is a framework for detecting likely failures in componentized systems. To detect failures, a Pinpoint server dynamically generates a model of the "good behavior" of the system. This good behavior is based on both the past behavior and the majority behavior of the system, under the assumption that most of the time, most of the system is likely to be behaving correctly.

When part of the system deviates from this believed good behavior, Pinpoint interprets the anomaly as a possible failure. Once Pinpoint notices a component generating a threshold number of these anomalies, Pinpoint triggers a restart of the component.

To detect failures in bricks, Pinpoint monitors each brick's vital statistics. Each brick sends its own statistics to the Pinpoint server at one-second intervals. Statistics are divided into *activity* and *state* statistics.

Activity statistics, e.g., the number of processed writes, represent the rate at which a brick is performing some activity. When Pinpoint receives an activity statistic, it compares it to the statistics of all the other bricks, looking for highly deviant rates. Because we want to be able to run SSM on a relatively small number of nodes, we calculate the median absolute deviation of the activity statistics. This

3.4 Load capacity discovery and admission control

In addition to the basic read/write algorithm, each stub maintains a sending window (SW) for each brick, which the stub uses to determine the maximum number of in-flight, non-acked requests the stub can send to the recipient brick.

The stub implements a additive-increase, multiplicative-decrease (AIMD) algorithm for maintaining the window; the window size is additively increased on a successful ack and multiplicatively decreased on a timeout. When a request times out, the stub reduces its sending window to the brick accordingly. In the case when the number of in-flight messages to a brick is equal to the SW, any subsequent requests to that brick will be disallowed until the number of in-flight messages for that brick is less than the SW. If a stub cannot find a suitable number of bricks to send the request to, it throws an exception to the caller indicating that the system is overloaded.

Each stub stores temporary state for only the requests that are awaiting responses from bricks. The stub performs no queueing for incoming requests from clients. For any request that cannot be serviced because of overload, the stub rejects the request *immediately*, throwing an exception indicating that the system is temporarily overloaded.

In addition, each brick performs admission control; when a request arrives at the brick, it is put in a queue. If the request timeout has elapsed by the time that the brick has dequeued the request, the request is disregarded and the brick continues to the service the next queued request.

Note that the windowing mechanism at the stub and the request rejection at the brick protect the system in two different ways. At the stub, the windowing mechanism prevents any given stub from saturating the bricks with requests. However, even with the windowing mechanism, it is still possible for multiple stubs to temporarily overwhelm a brick (e.g. the brick begins garbage collection and can no longer handle the previous load). At the brick, the request rejection mechanism allows the brick to throw away requests that have already timed out in order to "catch up" to the requests that can still be serviced in a timely manner.

3.5 Failure and Recovery

In SSM, recovery of any component that has failed is simple; a restart is all that is necessary to recover from a non-persistent failure. No special case recovery code is necessary.

On failure of a client, the user perceives the session as lost, e.g., if the browser crashes, a user does not necessarily expect to be able to resume his interaction with a web application. If cookies for the client are persisted, as is often the case, then the client may be able to resume his session when the browser is restarted.

On failure of a stateless application server, a restart of the server is sufficient for recovery. After restart, the stub on the server detects existing bricks from the beacons and can reconstruct the table of live bricks. The stub can immediately begin handling both read and write requests; to service a read request, the necessary metadata is provided by the client in the cookie. To service a write request, all that is required is a list of WQ live bricks.

On failure of a brick, a simple restart of the brick is sufficient for recovery. The contents of its memory are lost, but since each hash value is replicated on $WQ - 1$ other bricks, no data is lost. The next update of the session state will re-create WQ new copies; if $WQ - 1$ additional failures occur before then, data may be lost.

A side effect of having simple recovery is that clients, servers, and bricks can be added to a production system to increase capacity. For example, adding an extra brick to an already existing system is easy. Initially, the new brick will not service any read requests since it will not be in the read group for any requests. However, it will be included in new write groups because when the stub detects that a brick is alive, the brick becomes a candidate for a write. Over time, the new brick will receive an equal load of read/write traffic as the existing bricks, since load balancing is done per request and not per hash key.

3.6 Recovery Philosophy

Previous work has argued that rebooting is an appealing recovery strategy in cases where it can be made to work [6]: it is simple to understand and use, reclaims leaked resources, cleans up corrupted transient operating state, and returns the system to a known state. Even assuming a component is reboot-safe, in some cases multiple components may have to be rebooted to allow the system as a whole to continue operating; because inter-component interactions are not always fully known, deciding *which* components to reboot may be difficult. If the decision of which components to reboot is too conservative (too many components rebooted), recovery may take longer than really needed. If it is too lenient, the system as a whole may not recover, leading to the need for another recovery attempt, again resulting in wasted time.

By making recovery "free" in SSM, we largely eliminate the cost of being too conservative. If an SSM brick is *suspected* of being faulty – perhaps it is displaying fail-stutter behavior [2] or other characteristics associated with software aging [14] – there is essentially no penalty to reboot it prophylactically. This can be thought of as a special case of fault-model enforcement: treat any performance fault in an SSM brick as a crash fault, and recover accordingly. In recent terminology, SSM is a *crash-only* subsystem [6].

3.7 Brick MTTF vs. Availability

Before presenting experimental results, we illustrate the relationship between MTTF for an individual brick and the availability of data for SSM as a whole. We assume independent failures; when failures are correlated in Internet server clusters, it is often the result of a larger catastrophic failure which bricks are currently alive by listening to the beacons; stubs receive the announcements and make connections to the bricks via TCP/IP. We choose TCP/IP as the communication mechanism for read/write request traffic because reliable and ordered messaging enables easy prototyping.

When a stub contacts a brick, a stream is created between the two, which lasts until either component ceases executing. Each brick has a list of streams corresponding to the stubs that has contacted it. The brick has one main processing thread, which fetches requests from a shared inbox, and handles the request by manipulating the internal data structures. A single monitor thread handles the internal data structures. In addition, the brick has an additional thread for each stub communicating with the brick; each of these communication threads puts requests from the corresponding stub into the shared inbox.

The write function `Write(HashKey H, Object v, Expiry E)` exported by the stub returns a cookie if the write succeeds or throws an exception otherwise. The returned cookie is passed back to the client (Web browser) for storage, as it stores important metadata that will be necessary for the subsequent read. Existing solutions for session state also rely on storing this metadata on the client.

The read function `Read(Cookie C, HashKey H)` returns the most recently written value for hash key H, or throws an exception if the read fails. If a read/write returns to the application, then it means the operation was successful. On a read, SSM guarantees that the returned value is the most recently written value by the user.

The stub dispatches write and read requests to the bricks. Before we describe the algorithm describing the stub-to-brick interface, let us define a few variables. Call W the write group size. Call R the read group size. On a write request, a stub *attempts* to write to W of the bricks; on a read request, it attempts to read from R bricks.

Define WQ as the size of the write set, which is the minimum number of bricks that must return "success" to the stub before the stub returns to the caller. $WQ - 1$ is the number of simultaneous brick failures that the system can tolerate before possibly losing data. R is the size of the candidate read set; only 1 brick need to reply to service a read request succesfully. Note that $1 \leq WQ \leq W$ and $1 \leq R \leq WQ$. In practice, we use $W = 3, WQ = 2, R = 2$.

Lastly, call t the request timeout interval, the time that the stub waits for a brick to reply to an individual request, usually on the order of milliseconds. t is different from the session expiration, which is the lifetime of a session state object, usually on the order of minutes. We use t and *timeout* interchangeably in this paper. In practice, t is a rough upper bound on the time an application is willing to wait for the writing and retrieval of a session state object, usually on the order of tens to hundreds of milliseconds since session state manipulation is in the critical path of client requests.

3.2 Basic Read/Write Algorithm

The basic write algorithm can be described as "write to many, wait for a few to reply." Conceptually, the stub writes to more bricks than are necessary, namely W, and only waits for WQ bricks to reply. Sending to more bricks than are necessary allows us to harness redundancy to avoid performance coupling; a degraded brick will not slow down a request. In the case where WQ bricks do not reply within the timeout, the stub throws an exception so that the caller can handle the exception and act accordingly (e.g., signal to the end user to come back later), rather than being forced to wait indefinitely. This is part of the system applying backpressure. The algorithm is described below:

```
Cookie Write(HashKey H, Object v, Expiry E)
    throws SystemOverloadedException

Time wakeup = getCurrentTime() + timeout;
int cs = checksum(H, v, E);
Brick[] repliedBricks = {};
Brick[] targetBricks = chooseRandomBricks(W);
foreach brick in targetBricks
   do WriteBrick(H, v, E, cs);
while (repliedBricks.size < WQ)
   Time timeleft = wakeup - getCurrentTime();
   Brick replied = receiveReply(timeleft);
   if (replied == null) break;
   repliedBricks.add(replied);
if (repliedBricks.size < WQ)
   throw new SystemOverloadedException();
int check = checksum(H, repliedBricks, cs, E);
return new Cookie(check, H, repliedBricks, cs, E);
```

The stub handles a read by sending the read to R bricks, waiting for only 1 brick to reply:

```
Object Read(Cookie c) throws CookieCorruptedExcept,
   SystemOverloadedExcept, StateExpiredExcept,
   StateCorruptedExcept

int check = c.checksum;
int c2 = checksum(c.H, c.repliedbricks, c.cs, c.E);
if (c2 != check)
   throw new CookieCorruptedException();
if (isExpired(c.E))
   throw new StateExpiredException();
Brick[] targetBricks = c.repliedBricks;
foreach brick in targetBricks
  do RequestBrickRead(H, E, cs);

Brick replied = receiveReply(timeout);
if (replied == null)
   throw new SystemOverloadedException();
retval = replied.objectValue;
if (c.cs != checksum(retval))
   throw new StateCorruptedException();
return retval;
```

3.3 Garbage Collection

For garbage collection of bricks, we use a method seen in generational garbage collectors [7]. For simplicity, earlier we described each brick as having one hash table. In reality, it has a set of hash tables; each hash table has an expiration. A brick handles writes by putting state into the table with the closest expiration time after the state's expiration time. For a read, because the stub sends the key's expiration time, the brick knows which table to look in. When a table's expiration has elapsed, it is discarded, and a new one is added in its place with a new expiration.

so an advanced query mechanism to locate the user's state is unnecessary. Furthermore, the client is typically responsible for storing the necessary metadata to retrieve the state.
2. Is semi-persistent. Session state must be present for a fixed interval T, the application-specific session timeout (usually on the order of minutes to hours), but should expire after T.
3. Is written out in its entirety, and usually updated on every interaction.

Given these properties, the functionality necessary for a session state store can be greatly simplified, relative to fully-general ACID guarantees provided by a relational datbase. Each simplification corresponds to an entry in the previous numbered list:

1. No synchronization is needed. Since the access pattern corresponds to an access of a single user making serial requests, no conflicting accesses exist, and hence race conditions on state access are avoided, which implies that locking is not needed. In addition, a single-key lookup API is sufficient. Since state is keyed to a particular user and is usually only accessed by that user, a general query mechanism is not needed.
2. State stored by the repository need only be semi-persistent – a temporal, lease-like [16] guarantee is sufficient, rather than the durable-until-deleted guarantee that is made in ACID.
3. Atomic update is sufficient for correctness, since partial writes do not occur. Once session state is modified, any of its previous values may be discarded.

Relative to the specific requirements of session state, SSM does, in a sense, provide ACID guarantees: atomicity and bounded durability are provided, and consistency and isolation are made trivial by the access pattern.

As a generalization, the class of state that we address need not necessarily be single-user; as long as state ownership is explictly passed between parties, which is common in today's enterprise applications [21], the techniques discussed in this paper applies.

2.1 Existing Solutions

Frequently, enterprises use either a relational database (DB) or a filesystem or filesystem appliance (FS) to store session state, because they already use a DB or FS for persistent state. There are several drawbacks to using either a DB or FS, besides the costs of additional licenses, which are detailed in previous work [26].

In addition, DB and file systems are well-known to be difficult to administer and tune. Each must be configured and tuned for a particular workload. Even for a skilled and costly administrator, this remains a difficult and often error-prone process that is repeated as the workload changes.

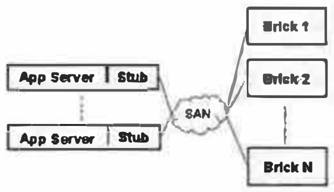


Figure 1: **Architecture of SSM. Stubs are stateless and are used by application servers to read and write state. Bricks are diskless components that store session state.**

In contrast, in-memory solutions avoid several of the drawbacks of FS/DB, and are generally faster, but make do not provide both performance and correctness guarantees. Existing in-memory solutions require a user to be pinned to a particular server, which prevents the application-processing tier from remaining truly stateless, since each server must both run application logic and store session state. Because of pinning, load-balancing can only be done across users but not across requests, and hotspots are harder to alleviate. A detailed discussion of in-memory solutions can be found in previous work [25, 26].

3. PROPOSED SOLUTION: SSM

We now describe the design and implementation of SSM, a lightweight session-state store. We make the following assumptions about the operating environment, which are typical of large-scale services [3]: a physically secure cluster interconnected by a commercially-available high-throughput, low-latency system area network (SAN); and an uninterruptible power supply to reduce the probability of a system-wide simultaneous hardware outages. The Java prototype consists of 872 semicolons and runs on the UC Berkeley Millennium Cluster, consisting of 42 IBM xSeries 330 1U rack-mounted PCs, each running Linux 2.4.18 on Dual 1.0 GHz Intel Pentium III CPUs and 1.5GB ECC PC133 SDRAM, connected via Gigabit Ethernet.

3.1 SSM Overview

SSM has two components: bricks and stubs. Bricks, each consisting of a CPU, network interface and RAM (no disk), provide storage; stubs dispatch read and write requests to bricks. Figure 1 shows the basic architecture of SSM.

On a client request, the application server will ask the stub to read the client's session state, and after application processing, to write out the new session state. The general strategy employed by the stub for both reads and writes is "send to many bricks, wait for few to reply," to avoid having a request depend on any *specific* brick. Upon completion of the write request, a cookie containing the ids of the bricks that processed the write is sent back to the client.

A brick stores session state objects using an in-memory hash table. Each brick sends out periodic multicast beacons to indicate that it is alive. Each stub keeps track of less tier" HA techniques for incremental scaling, fault tolerance, and overprovisioning.

2. We demonstrate the resulting simplicity of recovery management by combining SSM with a generic statistical-monitoring failure detection tool. Pinpoint looks for "anomalous" behaviors (based on historical performance or deviation from the performance of peer nodes) and immediately coerces any misbehaving node to crash and reboot. Although false positives do occur, the simplicity and low cost of recovery (crash and reboot) makes them a minor consideration, greatly simplifying SSM's failure detection and management strategy. Combined with SSM's additive increase/multiplicative decrease admission control that protects it from overload, the result is a largely self-managing subsystem using entirely generic detection and recovery techniques.
3. We summarize the design choices and lessons, along with the system architecture requirements that allow the approach to work, and highlight design principles that can be applied to other systems.

In Section 2, we define a category of session state, its associated workload, and existing solutions. In Section 3, we present the design and implementation of SSM, a recovery-friendly and self-managing session state store. In Section 4, we describe the integration of SSM with Pinpoint to enable the system to be self-healing. In Section 5, we present benchmarks demonstrating the features of SSM. In Section 6, we insert SSM into an existing production internet application and compare its performance, failure, and recovery characteristics with the original implementation. In Section 7, we discuss the design principles extracted from SSM. We then discuss related and future work, and conclude.

2. WHY SESSION STATE?

In networking systems, signaling systems for flow state [8] fall in between two extremes: hard-state and soft-state [32]. In hard-state systems, state is explicitly written once and remains written unless explicitly removed; special mechanisms exist to remove orphaned state. In contrast, in soft-state systems, state automatically expires unless refreshed by the writer, so no such special mechanisms are needed. Session state lies somewhere in between: unlike hard state, its maximum overall lifetime and inter-access interval are bounded, so persistence guarantees need only respect those bounds; unlike soft state, it cannot be reconstructed from other sources if lost, unless the user is asked to repeat all steps that led to the construction of the state.

Nearly all nontrivial Internet services maintain session state, but they either store it as hard state because that is what most storage systems provide, or store it ephemerally (in RAM of otherwise stateless components) because it is cheaper and faster. The former is overkill, the latter does not provide adequate guarantees of persistence, especially in the face of transient failures. Table 2 compares and contrasts the different types of state.

	Hard/Persistent	**Soft/Session**
Write Method	Write once	Refresh
Deletion Method	Explicit	Expiration
Orphan Cleanup	Manual	Automatic

Table 1: Key differences among hard, persistent, soft, and session state.

For the remainder of this paper, we will use the term "session state" to refer to the subcategory of user-session state we now describe. Many associate session state with "shopping cart," but the class of session state we address is significantly broader than just shopping carts. An example of application session state that we address includes user workflow state in enterprise applications. In particular, today's enterprise applications, such as those in J2EE, are often accessed via a web browser. All application state, such as context and workflow, is stored on the server and is an example of what we are calling session state. In essence, user workflow state in enterprise applications is equivalent to temporary application state on a desktop application. Another example of session state is travel itineraries from online travel sites, which capture choices that users have made during the shopping process. Shopping carts can also be an example of session state.

To understand how session state is typically used, we use the example of a user working on a web-based enterprise-scale application to illustrate the typical flow sequence. A large class of applications, including J2EE-based and web applications in general, use the interaction model below:

- User submits a request, and the request is routed to a stateless application server. This server is part of what is often called the middle-tier.
- Application server retrieves the full session state for user (which includes the current application state).
- Application server runs application logic
- Application server writes out entire (possibly modified) session state
- Results are returned to the user's browser

Session state is in the critical path of each interaction, since user context or workflow is stored in session state. Loss of session state is seen as an application failure to the end user, which is usually considered unacceptable to the service provider. Typical session state size is between 3K-200K bytes [37].

Some important properties/qualities of the session state we focus on are listed below. Session state:

1. Is accessed in a serial fashion by a single user (no concurrent access). Each user reads her own state, usually keyed by a deterministic function of the user's ID,

Session State: Beyond Soft State

Benjamin C. Ling, Emre Kıcıman and Armando Fox
{bling, emrek, fox}@cs.stanford.edu

ABSTRACT

The cost and complexity of administration of large systems has come to dominate their total cost of ownership. Stateless and soft-state components, e.g. Web servers or network routers, are easy to manage: capacity can be scaled incrementally by adding more nodes, rebalancing of load after failover is easy, and reactive or proactive ("rolling") reboots can be used to handle transient failures. We show that it is possible to achieve the same ease of management for the state-storage subsystem by subdividing persistent state according to the specific guarantees needed by each type. While other systems [19, 17] have addressed persistent-until-deleted state, we describe SSM, a store for a previously unaddressed class of state – user-session state – that exhibits the same manageability properties as stateless nodes while providing firm storage guarantees. Any node can be proactively or reactively rebooted at any time to recover from transient faults, without impacting online performance or losing data. We exploit this simplified manageability by pairing SSM with an application-generic, statistical-anomaly-based framework that detects crashes, hangs, and performance failures, and automatically attempts to recover from them by rebooting faulty nodes. Although the detection techniques generate some false positives, the cost of recovery is so low that the false positives have low impact. We provide microbenchmarks to demonstrate SSM's built-in overload protection, failure management and self-tuning. We benchmark SSM integrated into a production enterprise-scale interactive service to demonstrate that these benefits need not come at the cost of significantly decreased throughput or response time.

1. INTRODUCTION

The cost and complexity of administration of systems is now the dominant factor in total cost of ownership for both hardware and software.In addition, since human operator error is the source of a large fraction of outages [5], attention has recently been focused on simplifying and ultimately automating administration and management to reduce the impact of failures [13, 19], and where this is not fully possible, on building self-monitoring components [20]. However, fast, accurate detection of failures and recovery management remains difficult, and initiating recovery on "false alarms" often incurs an unacceptable performance penalty; even worse, initiating recovery on "false alarms" can cause incorrect system behavior when system invariants are violated [20].

Operators of both network infrastructure and interactive Internet services have come to appreciate the high-availability and maintainability advantages of stateless and soft-state [33] protocols and systems. The stateless Web server tier of a typical three-tier service [3] can be managed with a simple policy: misbehaving components can be reactively or proactively rebooted, which is fast since they typically perform no special-case recovery, or can be removed from service without affecting correctness. Further, since all instances of a particular type of stateless component are functionally equivalent, overprovisioning for load redirection [3] is easy to do, with the net result that both stateless and soft-state components can be overprovisioned by simple replication for high availability.

However, this simplicity does not extend to the stateful tiers. Persistent-state subsystems in their full generality, such as filesystem appliances and relational databases, do not typically enjoy the simplicity of using redundancy to provide failover capacity as well as to incrementally scale the system. We argue that the ability to use these HA techniques can in fact be realized if we subdivide "persistent state" into distinct categories based on durability and consistency requirements. This has in fact already been done for several large Internet services [31, 39, 28], because it allows individual subsystems to be optimized for performance, fault-tolerance, recovery, and ease-of-management.

In this paper, we make three main contributions:

1. We focus on *user session state*, which must persist for a bounded-length user session but can be discarded afterward. We show why this class of data is important, how its requirements are different from those for persistent state, and how to exploit its consistency and workload requirements to build a distributed, self-managing and recovery-friendly session state storage subsystem, SSM. SSM provides a probabilistic bounded-durability storage guarantee for such state. Like stateless or soft-state components, any node of SSM can be rebooted without warning and without compromising correctness or performance of the overall application. No node performs special-case recovery code. Additional redundancy allows multiple simultaneous failures. As a result, SSM can be managed using simple, "state-

sites demonstrate the performance, stability, and connectivity support of our proposed design. Additionally, this paper describes the construction of two Saxons-based overlay services.

It is conceivable for a common overlay structure management layer to allow runtime overhead sharing when overlay nodes host multiple services. However, different overlay services often desire different link density, structure qualities, and stability support. Additionally, although overlay groups may overlap, they often contain a substantially large number of non-overlapping nodes. These factors make it difficult for multiple services to share the same overlay structure. As a result, we believe it is more feasible for sharing low-level activities such as link property measurements. For instance, the discovery of a high bandwidth link to a particular node may interest multiple hosted services. Further investigation on this issue is needed in the future.

Acknowledgment: This work was supported in part by NSF grants CCR-0306473 and ITR/IIS-0312925. We would like to thank Yuan Sun, the URCS systems group, the anonymous referees, and our shepherd David Culler for their valuable comments. We are also indebted to Liudvikas Bukys and the PlanetLab support for making possible the wide-area experimentation in this study.

References

[1] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of the ACM SOSP*, pages 131–145, Banff, Canada, October 2001.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proc. of the ACM SIGCOMM*, pages 205–217, Pittsburgh, PA, August 2002.

[3] B. Bollobas. *Random Graphs.* Academic Press, London, UK, 1985.

[4] P. Brett. The PlanetLab support team, August 2003. Personal communication.

[5] R. L. Carter and M. E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Computer Science Department, Boston University, March 1996.

[6] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. In *Proc. of the FuDiCo Workshop*, Bertinoro, Italy, June 2002.

[7] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proc. of the ACM SIGMETRICS*, pages 1–12, Santa Clara, CA, June 2000.

[8] I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[9] M. Devera. Hierarchical token bucket. http://luxik.cdi.cz/~devik/qos/htb/.

[10] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *Proc. of the ACM SIGCOMM*, pages 43–56, Stockholm, Sweden, August 2000.

[11] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An Architecture for a Global Internet Host Distance Estimation Service. In *Proc. of the IEEE INFOCOM*, New York, NY, March 1999.

[12] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proc. of the ACM SIGCOMM*, pages 288–298, Boston, MA, September 1995.

[13] S. Hotz. *Routing Information Organization to Support Scalable Routing with Heterogeneous Path Requirements.* PhD thesis, Dept. of Computer Science, University of Southern California, 1994.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. of the USENIX OSDI*, San Diego, CA, October 2000.

[15] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proc. of the USENIX Symp. on Internet Technologies and Systems*, Seattle, WA, March 2003.

[16] Limeware. http://www.limeware.com.

[17] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. of the ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[18] E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-based Approaches. In *Proc. of the IEEE INFOCOM*, New York, NY, June 2002.

[19] National Laboratory for Applied Network Research. http://moat.nlanr.net/Routing/rawdata.

[20] Active Measurement Project at the National Laboratory for Applied Network Research. http://amp.nlanr.net.

[21] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. of the ACM SIGCOMM*, pages 139–152, Cannes, France, September 1997.

[22] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of the HotNets Workshop*, Princeton, NJ, October 2002.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM*, pages 161–172, San Diego, CA, August 2001.

[24] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. of the IEEE INFOCOM*, New York, NY., June 2002.

[25] University of Oregon Route Views Archive Project. http://archive.routeviews.org.

[26] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-End Effects of Internet Path Selection. In *Proc. of the ACM SIGCOMM*, pages 289–299, Cambridge, MA, August 1999.

[27] K. Shen. Distributed Hashtable on Pre-structured Overlay Networks. Technical Report TR831, Dept. of Computer Science, University of Rochester, January 2004. http://www.cs.rochester.edu/trs/systems-trs.html.

[28] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM*, pages 149–160, San Diego, CA, August 2001.

[29] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. IETF RFC-1075, November 1988.

[30] J. Winick and S. Jamin. Inet-3.0: Internet Topology Generator. Technical Report CSE-TR-456-02, Dept. of EECS, University of Michigan, 2002.

[31] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of the IEEE INFOCOM*, San Francisco, CA, March 1996.

[32] B. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark Routing on Overlay Networks. In *Proc. of the Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.

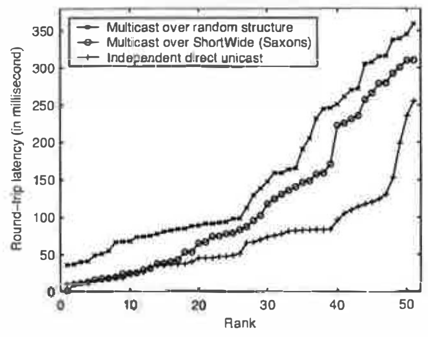


Figure 16: Multicast latency over 52 PlanetLab sites.

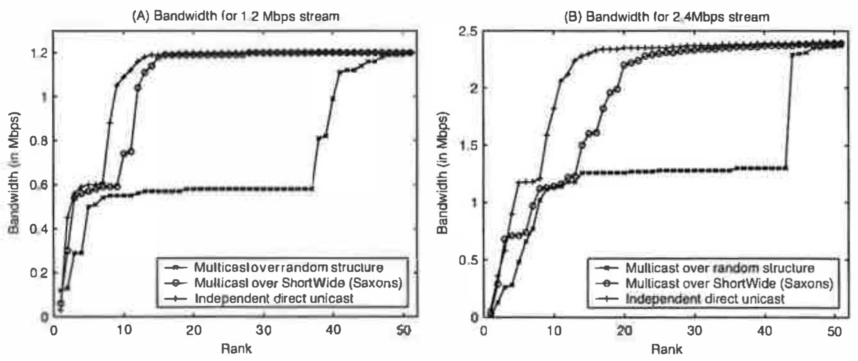


Figure 17: Multicast bandwidth on 52 PlanetLab sites.

creasingly on the bandwidth. We observe that the bandwidth performance of Saxons-based overlay multicast is quite close to that of the independent direct unicast for both 1.2Mbps and 2.4Mbps streams. Compared with multicast over random structure, the Saxons-based multicast provides near-loss-free ($< 5\%$ loss) data delivery to more than 4 times as many multicast receivers.

In addition to help the multicast service to achieve high performance, Saxons is also crucial for the scalability of this service. While it is conceivable for the multicast routing to run directly on the completely connected overlay, the overhead of tracking link properties and maintaining routing indexes along overlay links would become prohibitively expensive when the overlay grows to a large size. It should be noted that choosing an appropriate node degree range for the Saxons structure often has a significant impact on service performance. A highly connected Saxons structure makes the existence of high-performance overlay routes more likely. However, discovering them would consume more overhead at the service-level.

7 Related Work

The concept of structure-first overlay construction has been studied before in the Narada end-system multicast protocol [7]. Narada maintains a low latency mesh structure on top of which a DVMRP-style multicast routing protocol handles the data delivery. However, Narada is not designed for large-scale systems. For instance, its group management protocol requires each node to maintain the complete list of other overlay members, which would require excessive maintenance overhead for large overlays.

Prior studies have examined substrate-aware techniques in the construction of many Internet overlay services, including unicast overlay path selection (*e.g.*, RON [1]), end-system multicast protocols (*e.g.*, Overcast [14] and NICE [2]) and scalable DHT protocols (*e.g.*, Binning [24], Brocade [32], and Pastry [6]). These studies focused on specific services and substrate-aware techniques were often tightly integrated with the service construction. Saxons supports a comprehensive set of performance objectives in a separate overlay structure management layer and therefore it can be used as a building block to benefit the construction of a wide range of services. Additionally, we are unaware of any prior work on scalable overlay structure management that explicitly considers the overlay path latency, hop-count distance, and the overlay bandwidth at the same time.

Nakao *et al.* recently proposed a multi-tier overlay routing scheme [17]. In their approach, several overlay routing services are constructed on top of a topology probing kernel, which acquires AS-level Internet topology and routing information from nearby BGP routers. Saxons differs from their work by constructing the overlay structure using end-to-end network measurements. More importantly, their work does not explicitly address the multiple structure quality metrics that are investigated in this paper.

Many prior studies provided ideas that are related to the design of various Saxons components. For instance, a number of studies have examined scalable estimation schemes for finding nearby Internet hosts, including Hotz [13], IDMaps [11], GNP [18], and Binning [24]. While Saxons can utilize any of the existing techniques, we also introduce a light-weight random sampling approach to locate nearby hosts without the need of infrastructural support or established landmark hosts. Additionally, Kostić *et al.* recently proposed a random membership subset service for tree-shaped overlay structures [15]. This approach ensures that membership in the subset changes periodically and with uniform representation of all overlay nodes. The key difference with our random membership subset component is that Saxons is designed to support mesh-like overlay structures.

8 Concluding Remarks

In this paper, we propose Saxons, a substrate-aware overlay structure management layer that assists the construction of scalable Internet overlay services. Saxons dynamically maintains a high quality structure with low overlay latency, low hop-count distance, and high overlay bandwidth. At the same time, Saxons provides connectivity support to actively repair overlay partitions in the presence of frequent membership changes. Simulations and experiments on 55 PlanetLab the PlanetLab nodes are equipped with a packet filter [9] that limits the per-user outgoing bandwidth at 10Mbps [4]. With only 8 out of 55 nodes that are not subject to the bandwidth limit, ShortWide is able to provide high-speed overlay path (>10Mbps) for three times as many node pairs as its nearest competitor. This quantitative result is not typical due to the particular bandwidth control mechanism equipped on many of the PlanetLab nodes. Nonetheless, it provides an example of Saxons's ability to discover high bandwidth paths when they exist.

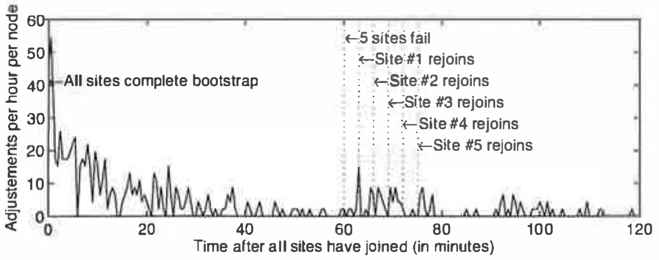


Figure 15: Saxons link adjustments on 55 PlanetLab sites.

Figure 15 shows the Saxons structure stability during membership changes. Again, nodes join the overlay network at the average rate of one per 3 seconds. We start tracking the link adjustment counts right after the last node has joined. Results are accumulated at 30-second intervals. We observe that the link adjustment rate mostly fall below 5 per hour per node after the 30th minute. We then inject a simultaneous 5-node failure at the 60th minute and we let them rejoin the overlay one by one at 3-minute intervals. We observe that the link adjustment activity is moderate (mostly under 10 per hour per node) during the membership changes. After the 100th minute, the average link adjustment count falls around 2 per hour per node, which indicates around one link adjustment at a single node during each 30-second interval.

6 Service Construction

Saxons actively maintains a stable and high quality overlay structure with partition repair support. Services can directly utilize the Saxons structure for overlay communication. Saxons can also benefit unicast or multicast overlay path selection services [1, 7, 14] by providing them a small link selection base, thus making them scalable without hurting their performance potential. In this section, we describe the construction of two services (query flooding and overlay multicast) that utilize the Saxons overlay structure in different ways. We have also implemented a Saxons-based distributed hash table and compared its performance against a well-known DHT protocol. Results of that work are reported in [27].

Saxons-based query flooding. We implemented the Gnutella query flooding protocol on top of the Saxons structure management layer. The default Gnutella protocol uses a bounded-degree random structure, which is similar to the *Random* structure we used in our evaluations. Our service directly uses the Saxons structure for query flooding without any further link selection. It uses the Saxons direct link query interface instead of the callback interface because no link-related state is actively maintained at the service-level. The low overlay hop-count distance in the Saxons structure allows query flooding to reach more nodes at a particular TTL. The low overlay latency allows fast query response while the high overlay bandwidth alleviates the high network load of query flooding. We do not provide performance results in this paper because they closely match the raw Saxons performance shown earlier.

Saxons-based overlay multicast. We also implemented an overlay multicast routing service on Saxons. Similar to DVMRP [29], our multicast service is based on *Reverse Path Forwarding* over a distance vector unicast routing protocol. The service actively monitors link latency and bandwidth between Saxons neighbors while the DV unicast routing protocol maintains path latency and bandwidth by aggregating the link properties. We employ a simple path cost function in the DV protocol that considers path latency, bandwidth, and hop-count distance: $cost = \frac{latency}{L_{unit}} + \frac{hop}{H_{unit}} - \frac{bandwidth}{B_{unit}}$. We empirically choose L_{unit}=20ms, H_{unit}=1; and B_{unit}=1.0Mbps in our implementation. Note that our main purpose is to demonstrate the effectiveness of Saxons in supporting overlay service construction. Therefore we do not pursue optimized service implementation in this paper.

We evaluated the performance of the Saxons-based overlay multicast routing service on the PlanetLab testbed. We choose a PlanetLab node that does not have outgoing bandwidth control as the multicast source: `planetlab2.cs.duke.edu`. The overlay structure is configured at the node degree range of <4−12>. We compare the performance of Saxons-based overlay multicast with the same multicast service running on top of a random overlay structure. For additional comparison purposes, we approximate the *ideal* multicast performance using the independent direct unicast, which measures the latency and bandwidth along the direct Internet path from the source to each receiver in the absence of simultaneous traffic. Figure 16 illustrates the round-trip latency from the source to all receivers, ranked increasingly on the latency. We observe that Saxons-based overlay multicast achieves 24% less latency in average than multicast over random structure. Compared with independent direct unicast, overlay multicast produces longer latency due to its multi-hop forwarding nature.

We also measured the multicast bandwidth on the PlanetLab testbed. We conducted two experiments, one with a 1.2Mbps multicast stream and another with a more aggressive 2.4Mbps stream. For unicast transport along overlay links, we use the default UDP service without lost recovery or congestion control. Congestion-controlled transport protocols such that TFRC [10] may improve the performance of our service implementation. However, it is not crucial to our purpose of evaluating the Saxons overlay structure management. Figure 17 shows observed bandwidth at all receivers, ranked in-

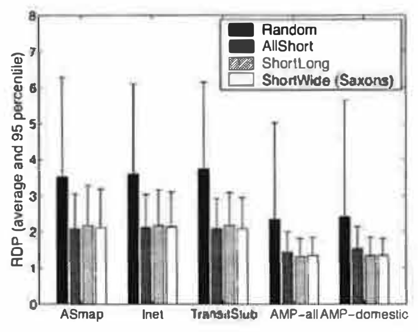


Figure 10: Backbone topologies.

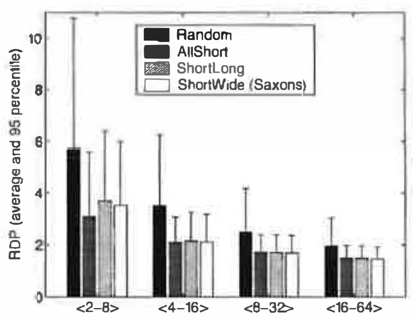


Figure 11: Node degree ranges.

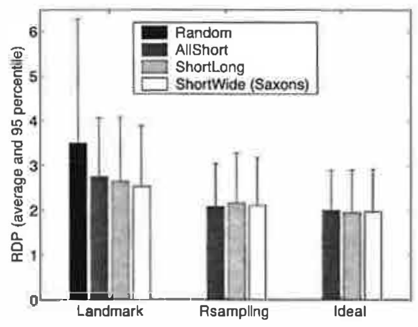


Figure 12: Finding nearby hosts.

linked and run inside the application process space. A standalone Saxons process allows possible runtime overhead sharing when overlay nodes host multiple services.

```
/* type def. of link adjustment callback functions */
typedef int (*SX_CALLBACK)(int linkcnt, link_t *links);

/* join an overlay group */
int sx_join(char *olgroup, int activedegree,
            int maxdegree, SX_CALLBACK cb_adjustment);

/* leave an overlay group */
int sx_leave(char *olgroup);

/* get directly attached overlay links */
int sx_getlinks(int *ptr_linkcnt, link_t *links);
```

Figure 13: The core C/C++ API for Saxons.

Figure 13 shows the core C/C++ interface for developing overlay applications on Saxons. In particular, we provide two ways for overlay applications to access the structure information in Saxons. First, they can directly query the Saxons layer to acquire information about attached overlay links (`sx_getlinks`). They can also provide a nonblocking callback function[2] (`cb_adjustment`) at the startup. If so, Saxons will invoke the application-supplied callback function each time an overlay link adjustment occurs. Link adjustment callbacks are useful for applications that maintain link-related state, such as various overlay routing services. Without the callback mechanism, they would have to poll the Saxons layer continuously to keep their link-related state up-to-date.

We conducted experiments on the PlanetLab testbed [22] to evaluate the Saxons performance in a real-world environment. We compare the overlay structure quality achieved by various quality maintenance policies: Random, AllShort, ShortLong, and ShortWide. In the experiments, nodes join the overlay network at the average rate of one per 3 seconds. Measurement results are taken when the average link adjustment rate falls below one per hour per node. Overlay structures are configured with the node degree range of <4−16>. Figure 14 illustrates the Saxons overlay latency and bandwidth CDFs for all node pairs among 55 PlanetLab nodes, all from unique wide-area sites. We provide round-trip latency results because

[2]Callback functions are not allowed to block on I/Os and they must return in a bounded amount of time.

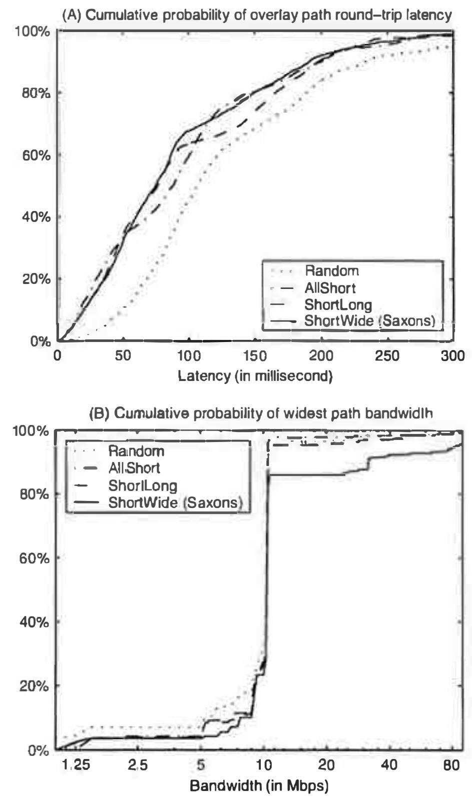


Figure 14: Saxons overlay latency and bandwidth CDFs for all site pairs among 55 PlanetLab sites. The X-axis for Figure (B) is in the log scale.

they are easier to measure than one-way latencies. Note that the simulation results shown earlier are for one-way latencies. We do not show the performance on the overlay hop-count distance because the overlay size is too small to make this metric meaningful.

In terms of overlay latency, all three quality maintenance policies outperform the random overlay structure with over 18% less overlay path latency in average. This performance difference is close to the simulation result for small overlays in Figure 6(A). As for the bandwidth, we observe that most of the node pairs have 10Mbps overlay bandwidth between them. As discussed in Section 3.2, this is because most of

Average node lifetime	Connectivity	Relative delay penalty	Hop-count distance	Bandwidth along widest paths	Average per-node link adjustments	
					Partition repair	Quality maintenance
7.5 minutes	96.0%	4.13	6.13	18.08 Mbps	0.24 links/hour	17.04 links/hour
15 minutes	99.1%	3.93	6.03	19.57 Mbps	0.07 links/hour	11.13 links/hour
30 minutes	100.0%	3.76	6.02	21.50 Mbps	0.02 links/hour	6.85 links/hour
1 hour	100.0%	3.55	5.99	22.60 Mbps	0.01 links/hour	4.08 links/hour
2 hours	100.0%	3.49	5.99	24.09 Mbps	0.00 links/hour	2.34 links/hour
Infinity	100.0%	3.41	5.97	26.73 Mbps	0.00 links/hour	0.00 links/hour

Table 4: Stability and connectivity under frequent node joins and departures (3200-node overlay).

adjustment rate in Figures 9(C). The RDP and overlay bandwidth values are sampled at 30-second intervals and the link adjustment counts are accumulated at the same frequency. We observe that most of the link adjustments occur in the first 20 minutes. Further adjustments occur at very low rate (less than 2 links/hour per node), but they are crucial in continuously optimizing the structure quality in terms of overlay latency and bandwidth. We do not show the hop-count distance stabilization which stays mostly the same over the whole period. This is because the random structure generated by node bootstraps already has a low overlay hop-count distance.

Table 4 illustrates the Saxons stability and connectivity support under frequent node joins and departures at various membership change rates for 3200-node overlays. The partition repair scheduling delay parameters are set as D_l=0.5 and D_u=4.0 (defined in Section 3.6). Individual node life times are picked following the exponential distribution with the proper mean. We include results at some unrealistically high rates of membership changes (*e.g.*, average node lifetime of 7.5 minuets) to assess the worst case performance. For the same reason, we use a relatively sparse overlay structure with the node degree range <2−8>. Therefore, performance values here is not directly comparable with results shown earlier. The connectivity values in Table 4 are the percentage of fully connected network snapshots out of 5,000 samples. Other values are the average of samples taken at 30-second intervals. We observe that the Saxons connectivity support keeps the overlay structure mostly connected even under highly frequent overlay membership changes. Furthermore, this connectivity support is provided at a very low rate of link adjustments (up to 0.24 links/hour per node). We also observe that the structure quality degrades gracefully as the overlay membership changes become more frequent. Such structure quality maintenance incurs a moderate link adjustment rate of up to 6.85 links/hour for average node lifetime of 30 minutes or longer.

4.4 Impact of Factors

We study the performance impact of backbone topologies, node degree ranges, and the scheme for finding nearby hosts. All results shown in this section are for 3200-node overlays. Figure 10 illustrates the impact of different backbone topologies on the average and 95 percentile overlay RDP. We observe that the performance results are largely stable with different backbone topologies. Overlay RDPs are lower for AMP topologies due to their small size. Very similar results are found for overlay bandwidth and hop-count distance. We do not show them here due to the space limitation.

Figure 11 shows the impact of node degree ranges on the overlay RDP. We observe that overlays with higher link densities tend to make the existence of high-quality paths more likely. However, the relative performance difference among various overlay structure construction approaches remain mostly unchanged. Again, this conclusion is also true for overlay bandwidth and hop-count distance.

Figure 12 shows the overlay RDP for different structure construction schemes under Rsampling and the landmark-based Cartesian distance approach. The Landmark approach uses 8 landmarks in this experiment. We also show the results for an *Ideal* case where the actually closest host is always chosen. The result for Random is only shown for the Landmark approach since its performance is not affected by the policy for finding nearby hosts. We observe that Rsampling constructs structures with less overlay latency compared with the Landmark approach. In particular, it achieves 24% less overlay RDP for AllShort. This is because Rsampling can gradually converge to the optimal selection when running repeatedly. Though such performance is achieved at the cost of substantially more link adjustments during stabilization, we believe the benefit of higher-quality connectivity structure would offset such cost in the long run. Rsampling still generates around 10% higher RDP than Ideal since link adjustments stop when new links are no better than existing links over the specified threshold.

5 Implementation and Experimentation on PlanetLab

We have made a prototype implementation of the Saxons overlay structure management layer. Our prototype assumes the availability of a DNS-like naming system that maps each overlay group name to a small number of bootstrap nodes in a round-robin fashion. Most of the Saxons components are implemented in a single event-driven daemon while the network measurement daemon runs separately due to its time-sensitive nature. Our Saxons prototype can run as a standalone process communicating through UNIX domain sockets with hosted overlay applications linked with a Saxons stub library. Alternatively, the whole Saxons runtime can be dynamically

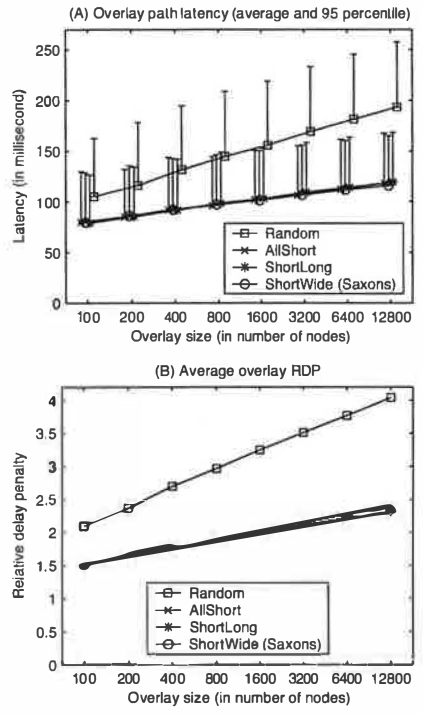


Figure 6: Overlay latency.

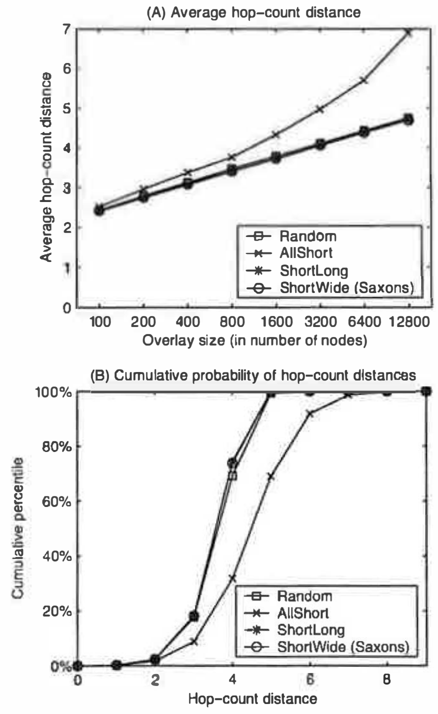


Figure 7: Hop-count distance.

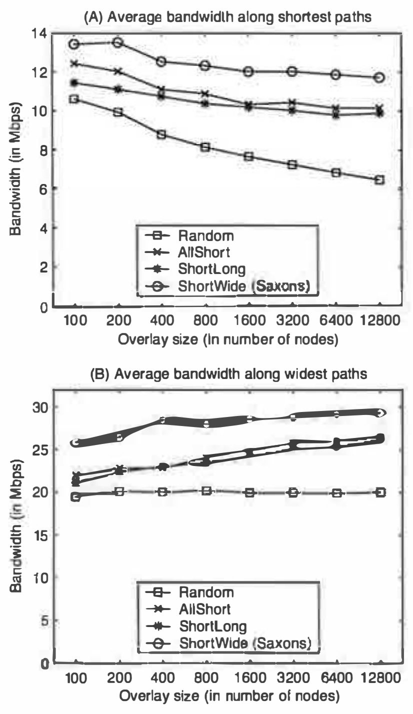


Figure 8: Overlay bandwidth.

other schemes, with around 47% larger average overlay hop-count distance for 12800-node overlays. All other three approaches perform very close in the hop-count distance due to the employment of randomness in the overlay structure construction [3].

Overlay bandwidth. Figure 8 shows the average overlay bandwidth along shortest paths and widest paths. We observe that ShortWide outperforms other approaches in terms of overlay bandwidth. For 12800-node overlays, the improvement over its nearest competitor is 15% for the bandwidth along shortest paths and 12% for the bandwidth along widest paths. We also observe that both AllShort and ShortLong significantly outperform Random in overlay bandwidth. This is because short overlay paths often have high bandwidth. Such an inverse correlation between latency and bandwidth is even stronger for TCP-based data transfer due to its various control mechanisms.

In summary, the ShortWide structure management policy outperforms other policies in terms of overlay path bandwidth while achieving competitive performance in terms of overlay path latency and hop-count distance. The results also confirm our conjecture that the AllShort policy could produce overlay structures with high hop-count distances.

4.3 Stability and Connectivity

In this section, we first study the system stabilization at the startup, *i.e.*, right after a large number of nodes have joined the overlay. We then examine Saxons's stability and connectivity support under frequent node joins and departures.

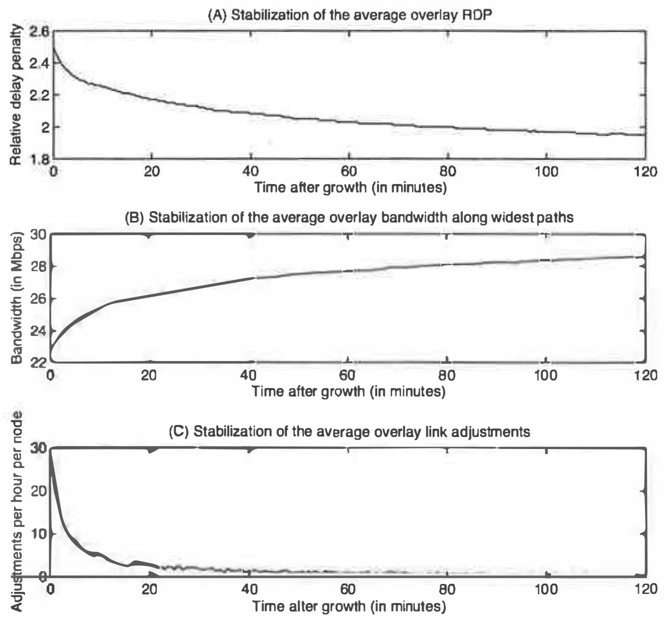


Figure 9: Stabilization after growth (3200-node overlay).

Figure 9 shows the system stabilization at the overlay startup. In this simulation, nodes join the network at the average rate of 10 joins/second with exponentially distributed inter-arrival time. Node joins stop when the desired overlay size is reached and we measure network samples after this point to assess the system stabilization. Figures 9(A) and 9(B) illustrate the stabilization of average RDP and average overlay bandwidth respectively. We also show the average link

Backbone	Node count	Link latency
ASmap	3,104	1−40ms
Inet	3,050	1−40ms
TransitStub	3,040	1−20ms for stub links 1−40ms for other links
AMP-all	118	measurement
AMP-domestic	108	measurement

Table 3: Backbone networks. A random bandwidth between 1.5−45Mbps and 45−155Mbps (with 50% probability for each range) is assigned for each backbone link.

ture quality maintenance and partition repairs. We will evaluate such link adjustment overhead in Section 4.3.

4 Simulation Results

Our performance evaluation consists of simulations (this section) and Internet experiments (Section 5). The goal of simulation studies is to assess the effectiveness of proposed techniques for large-scale overlays while Internet experiments illustrate the system performance under particular real-world environments.

4.1 Simulation Methodology and Setup

We use a locally-developed discrete-event simulator in our evaluations. We simulate all packet-level events at overlay nodes. We do not simulate the packet routing at the substrate network routers. Instead, we assume shortest-path routing in the substrate network and use that to determine the overlay link latency and bandwidth. We acknowledge that this model does not capture packet queuing delays or packet losses at routers and physical links. However, such a tradeoff is important to allow us achieve reasonable simulation speed for large networks.

The substrate networks we use in the simulations are based on four sets of backbone networks including a measurement-based one. First, we use Internet *Autonomous Systems* maps extracted from BGP routing table dumps, available at NLANR [19] and at the Route Views Archive [25]. Second, we include some transit-stub topologies generated using the GT-ITM toolkit [31]. We also use topologies generated by the Michigan *Inet*-3.0 Internet Topology Generator [30]. For ASmap and Inet topologies, we assign a random link latency of 1−40ms. For TransitStub topologies, we assign a random link latency of 1−20ms for stub links and 1−40ms for other links. Our final set of backbone network is based on end-to-end latency measurement data among 118 Internet nodes, reported by the NLANR Active Measurement Project [20]. Table 3 lists some specific backbone networks we used in our evaluations. The AMP-domestic network excludes 10 foreign hosts from the full AMP dataset. These 10 hosts have substantially larger latencies to other hosts than the average host-to-host latency. With a given backbone network, each overlay node in our simulations is randomly attached to a backbone node through an edge link. We assign a random latency of 1−4ms for all edge links. In terms of link bandwidth, a random bandwidth between 1.5−45Mbps and 45−155Mbps (with 50% probability for each range) is assigned for each backbone link. Edge links are assigned 100Mbps. These reflect commonly used T1, T3, OC-3, and Ethernet links.

The evaluation results are affected by many factors, including the substrate network topologies, protocol parameters, and the combination of different schemes for various Saxons components. Our strategy is to first demonstrate the effectiveness of proposed techniques at a typical setting and then explicitly evaluate the impact of various factors. Unless stated otherwise, results in Sections 4.2 and 4.3 are all based on the ASmap backbone topology, the Rsampling scheme for finding nearby hosts, and a node degree range of <4−16> (*i.e.*, d_a=4 and d_t=16). All periodic Saxons routines run at 30-second intervals except for bandwidth-oriented structure adjustments, which run at 120-second intervals. The link adjustment threshold for the structure quality maintenance is max{4ms, 10%} for short links (*i.e.*, a new link replaces an old link when it is at least 4ms and 10% shorter in latency) and max{1.0Mbps, 20%} for wide links. All other protocol parameters default to those described in Section 3.7.

4.2 Structure Quality

We compare different quality maintenance approaches in constructing high-quality overlay structure. AllShort represents protocols optimized solely for low overlay latency. ShortLong introduces a certain degree of randomness to the overlay structure. ShortWide is designed for achieving low overlay latency, low hop-count distance, and high overlay bandwidth at the same time. We also include a *Random* approach in our comparison. This approach makes no quality-oriented link adjustment after randomly establishing links during bootstrap.

Overlay latency. Figure 6 illustrates the structure quality on overlay latency at different overlay sizes. For each overlay size, nodes join the network at the average rate of 10 joins/second with exponentially distributed inter-arrival time. Node joins stop when the desired overlay size is reached and the measurement results are taken after the system stabilizes, *i.e.*, when the average link adjustment rate falls below one per hour per node. Figure 6(A) and 6(B) show the results in overlay path latency and the relative delay penalty respectively. We show both the average values and the 95 percentile values for the overlay path latency results. 95 percentile values do not exhibit different patterns from the average values and we do not show them in other figures for clarity. Overall, we observe all three schemes perform significantly better than the random overlay construction, especially for large networks.

Hop-count distance. Figure 7(A) shows the results on structure hop-count distance averaged over all node pairs at different overlay sizes. Figure 7(B) illustrates the cumulative probability of all-pair hop-count distances for 3200-node overlays. We observe that AllShort performs much worse than

It should be noted that latency and bandwidth measurements may not be always accurate. In order to avoid link oscillations, we require that a link adjustment occurs only when the new link is shorter or wider than the existing overlay link for more than a specified threshold.

3.6 Connectivity Support

In large-scale self-organizing overlay services, the fault or departure of a few *bridging* nodes may cause the partition of remaining nodes. Without careful consideration, the structure quality maintenance may also create overlay partition by cutting some *bridging* links. Saxons provides overlay connectivity support that actively checks the overlay connectivity and repairs partitions when they occur.

The Saxons connectivity support is based on periodic broadcasts of sequenced connectivity messages from a core node C. Let t_{int} be the interval between consecutive broadcasts. The connectivity messages flood the network along the overlay links. Node A detects a possible partition when the connectivity message is not heard for $t_{int} + t_{A,C}$, where $t_{A,C}$ is A's estimate of the message propagation delay upper-bound from C. When a partition is detected, A schedules a repair procedure at a random delay chosen uniformly from $[D_l t_{int}, D_u t_{int}]$. In the repair procedure, the node randomly picks another node from its local random-subset and attempts to establish a partition repair link. This procedure may fail because the contacted node may have reached its degree bound, be disconnected too, or have departed from the system. The partition repair procedure is continuously rescheduled at random delays until the connectivity messages are heard again. While it is always possible to reconnect to the network by directly establishing a link to the core node, this should be avoided since the core could be inundated with such requests. Nondeterministic delays in scheduling repair procedures are important to avoid all nodes in the partitioned network initiate such repair simultaneously. In many cases, successful repair at a single node can bring the network completely connected again.

In addition to partition repairs, Saxons also tries to avoid partitioning caused by link adjustments of the structure quality maintenance. This is achieved by having each node remember its upstream link to the core, defined as the link through which the connectivity message bearing the highest sequence number first arrived. The structure quality maintenance can avoid causing overlay partition by preserving the upstream link to the core.

The availability of the core node is critical to the Saxons connectivity support. In the case of the core node failure or physical disconnection from the network, no overlay nodes would succeed in its regular repair procedure. At several repeated failures, a node will ping the core node to determine whether a physical disconnection occurs. If so, the node then waits for a random delay before trying to broadcast connectivity messages as the new core node. Simultaneous broadcasts are arbitrated based on a deterministic total order among all nodes, *e.g.*, ordering by IP addresses. The same arbitration applies when multiple disconnected partitions rejoin with a core in each partition.

Saxons component	Overhead per interval	
	Active overhead	Passive overhead
Membership	$\leq d_t$ msgs	$\leq d_t$ msgs
Connectivity	$\leq d_t$ msgs	$\leq d_t$ msgs
Latency meas.	$f_{rs} * N_l$ pings	$\sim f_{rs} * N_l$ pings
Bandwidth meas.	$3 * N_b * S_b$	$\sim 3 * N_b * S_b$

Table 2: Saxons overhead. d_t is the node degree bound and f_{rs} is the random sampling factor. N_l and N_b are message counts for the latency and bandwidth measurements respectively. S_b denotes the message size for bandwidth measurements.

3.7 System Overhead

Table 2 illustrates the per-interval overhead of the Saxons components under stable conditions. Note that different Saxons components can run at different intervals. An overhead is counted as *active* when the node in question initiates the network transmission. Both the latency and bandwidth measurements are part of the ShortWide structure quality maintenance policy. Below we attempt to quantify the system overhead in a typical setting. We separate the overhead of bandwidth measurements from other overhead to highlight its dominance in resource consumption. The connectivity messages and the latency measurement messages are small (8 bytes each) in our implementation. A membership message with 20 member records at 8 bytes each[1] has a size of 160 bytes. Assume all these components run at 30-second intervals, the node degree bound d_t=16, the random sampling factor f_{rs}=4, and the latency measurement message count N_l=10. Accounting for the 28-byte IP and UDP headers, the Saxons runtime overhead excluding bandwidth measurements is about 1.3Kbps.

Bandwidth measurements are typically run at a low frequency, *e.g.*, 120-second intervals. Assume the message count N_b=20 and message size S_b=8KB, the bandwidth measurement overhead is about 32Kbps. The total Saxons management cost under this protocol setting is similar to the RON probing overhead for a 50-node overlay [1]. Note that the Saxons overhead does not directly depend on the overlay size, and thus it is able to achieve high scalability. Since most of the network overhead is caused by the bandwidth measurement, nodes that cannot afford such overhead can reduce the frequency of bandwidth measurements or even disable it. This would simply result in lower performance in overlay bandwidth. Bandwidth-oriented structure maintenance can also be made more efficient by adjusting the interval between consecutive runs depending on the system stability. For instance, it can run less often when prior runs result in no link adjustments, an indication that the overlay structure has stabilized.

During service growth or frequent membership changes, additional overhead of link adjustments is incurred for struc-

[1]Each membership record contains a 4-byte IPv4 address and a 4-byte timestamp.

Systems map available at NLANR [19]. More details for this network map and the simulation setup will be described in Section 4.1. The metric *latency stretch* in Figure 5 is defined as the ratio of the latency to the selected host to the latency to the optimal (*i.e.*, the closest) host. The Rsampling approach tests the network latency to four randomly selected nodes at each run and the old selection is replaced if a newly tested node is closer. The performance for Rsampling after 1, 4, and 16 runs are shown, compared with the performance of the landmark approach with 4, 8, and 16 landmark nodes. We observe that the Rsampling performance after 4 runs is competitive to the landmark approach even for 12800-node overlays. This performance is achieved with only a total of $4 \times 4 = 16$ latency tests.

Note that the random sampling technique could also be used for finding hosts with high bandwidth connections. The main difference is that bandwidth measurements cannot be conducted frequently because they are much more expensive than latency tests.

3.4 Node Bootstrap

When a node joins the Saxons connectivity structure, it must first know at least one active bootstrap node through out-of-band means. In principle, every active overlay member can serve as a bootstrap node. However, employing a small number of bootstrap nodes for each overlay group allows the use of a DNS-like naming system to locate them. Since joining nodes do not establish direct links to bootstrap nodes in our scheme (described later), it is feasible to employ a small number of bootstrap nodes as long as they are not overloaded with processing bootstrap requests and a desired level of availability can be provided.

During bootstrap, the joining node first contacts an bootstrap node to acquire a list of nodes randomly selected from the bootstrap node's local random-subset. The joining node then attempts to establish links with d_a nodes in the list. Link establishment attempts may fail because a target node may have departed from the system or have reached its degree bound. Further attempts may be needed to complete link establishments. As soon as the initial links are established, the joining node starts the random membership subset component to learn the existence of other nodes and also makes itself known to others. Periodic structure quality maintenance and connectivity management routines are also scheduled after the bootstrap.

3.5 Structure Quality Maintenance

A main goal of the Saxons structure management is to continuously maintain a high-quality overlay structure connecting member nodes. The structure quality is determined along three lines: low overlay latency, low hop-count distance, and high overlay bandwidth. The structure management component runs at a certain link density, specified by a node degree range $<d_a - d_t>$. Each node can initiate the establishment of d_a overlay links (called *active* links) and each node also passively accepts a number of link establishments (called *passive* links) as long as the total degree does not exceeds d_t. The degree upper-bound is maintained to control the stress on each node's physical access link and limit the impact of a node failure on the Saxons structure. Note that the average node degree is $2d_a$ under such a scheme. Below we consider several different approaches to maintain the structure quality. In all cases, a routine is run periodically to adjust active links for potentially better structure quality.

The overlay structure quality maintenance has been studied in the context of end-system multicast. The Narada protocol maintains a low-latency structure through greedily maximizing a latency-oriented utility value [7]. This approach requires a complete membership view at each node and a full-scale shortest-path routing protocol running on the overlay network, which may not be feasible for large-scale services. We include a more scalable latency-only approach in our study. This approach, called *AllShort*, continuously adjusts active links to connect to closest hosts it could find. More specifically, it utilizes the random sampling policy to measure the latency to a few randomly selected hosts and replace the longest existing active links if new hosts are closer. Note that such link adjustments must not violate rules in the Saxons connectivity support described in the next section.

Latency-only protocols like AllShort tend to create mesh structures with large hop-count distances. Consider a two-dimensional space with uniform node density and assume the network latency is proportional to the Cartesian distance. Let n be the total number of nodes and assume the node degree is bounded by a constant. Latency-only protocols would create grid-like structures with the hop-count diameter of $O(\sqrt{n})$, much larger than the $O(ln(n))$ diameter for randomly connected structures [3]. A straightforward idea is to add some random links into the structure to reduce the overlay hop-count distance. The second approach we consider, called *ShortLong*, was proposed by Ratnasamy *et al* [24]. In this approach, each node picks $d_a/2$ neighbors closest to itself and chooses the other $d_a/2$ neighbors at random (called *long links*).

However, neither of the above schemes considers the overlay bandwidth. To this end, we propose the third approach, called *ShortWide*, that also optimizes the overlay structure for high bandwidth. In this approach, half of the active links are still maintained for connecting to closest hosts each node could find. The other half are connected to randomly chosen hosts with high bandwidth (we call these *wide links*). The wide links are also maintained using random sampling, although at a much lower adjustment frequency and higher adjustment threshold than latency-oriented link adjustments. These are made necessary by the high overhead and inaccuracy of bandwidth measurements. Additionally, the high adjustment threshold preserves a large amount of *randomness* in the overlay structure, which is important for achieving low overlay hop-count distance.

the last messages (denoted by t_{first} and t_{last}). Note that A may not receive all messages due to congestion and drops at buffer queues. Assume A actually received $\tilde{N}_b$ messages, we determine the link bandwidth as $(\tilde{N}_b - 1) * S_b/(t_{last} - t_{first})$. In order to avoid transient network congestions, we repeat the tests three times with a random interval between 2 and 6 seconds and take the median value from the three rounds as the final result.

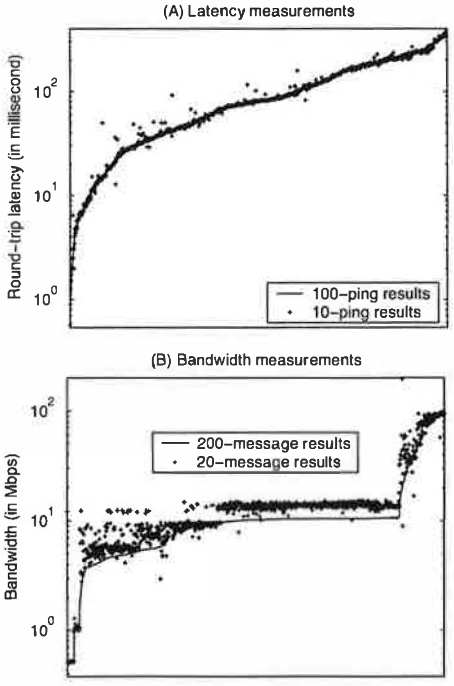


Figure 4: Network measurement results for all-to-all site pairs on 61 PlanetLab sites. Y-axises are in the log scale.

In practice, we use 10 pings for latency measurements (*i.e.*, N_l=10) and we use 20 UDP messages of 8KB for each of the three bandwidth measurement rounds (*i.e.*, N_b=20 and S_b=8KB). Each bandwidth test costs 480KB at this setting. We conducted experiments on the PlanetLab testbed [22] to assess the effectiveness of our network measurement schemes. In the experiments, we compare our measurement results at the above mentioned setting with results using 10 times more messages. Figure 4 illustrates the measurement results for all-to-all node pairs between 61 PlanetLab nodes, all from unique wide-area sites. For both latency and bandwidth measurements, the results are ranked in ascending order for the more accurate measurements that use 10 times more messages. Figure 4(A) shows that the latency measurement with 10 pings are already very accurate. From Figure 4(B), we notice that a large number of site pairs have 10Mbps bandwidth between them. It turns out that many PlanetLab nodes are equipped with the *Hierarchical Token Bucket* filter [9] that limits the per-user outgoing bandwidth at 10Mbps [4]. Our measurements give slightly higher bandwidth estimates for these links. It appears the reason is that these filters let go about 64KB data before the rate control kicks in.

We should point out that almost any network measurement techniques can be used in Saxons. And different schemes may be better suited for different network environments. For instance, the behavior of the Hierarchical Token Bucket filter requires the bandwidth measurement to use much larger than 64KB data for being effective.

3.3 Finding Nearby Hosts

In addition to network performance measurement, finding nearby hosts is also needed by the Saxons overlay structure management. Accuracy, scalability, and ease of deployment are some important issues for this component. Previous studies have proposed various techniques for locating nearby hosts [11, 12, 18, 24], most of which require infrastructure support or established landmark hosts. Saxons can utilize any of the existing techniques in principle. For ease of deployment, we introduce a random sampling approach that does not require any infrastructure support or landmark hosts.

The basic idea of random sampling, or *Rsampling*, is to randomly test the network latency to f_{rs} (or the random sampling factor) nodes from the overlay group and picks the one with shortest latency. The overhead of this approach can be controlled by choosing a small f_{rs}. The performance of Rsampling is not directly competitive to more sophisticated landmark-based schemes. However, in addition to its advantage of ease of deployment, it has the property of converging to the closest host when running repeatedly. We compare Rsampling with the *landmark-based Cartesian distance* approach for locating nearby hosts. This approach requires a set of l well-known landmark hosts spread across the network and each landmark defines an axis in an l-dimensional Cartesian space. Each group member measures its latencies to these landmarks and the l-element latency vector represents its coordinates in the Cartesian space. For nearby host selection, a node chooses the one to which its Cartesian distance is minimum. This approach has been shown to be competitive to other landmark-based schemes [24].

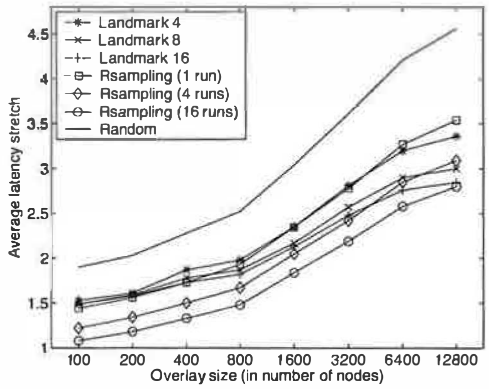


Figure 5: Performance of schemes for finding nearby hosts.

Figure 5 illustrates the simulation performance of latency estimation schemes. The backbone substrate network used in this experiment is based on a 3104-node Internet *Autonomous*

3.1 Random Membership Subsets

Accurately tracking the complete list of group members often requires the dissemination of such complete list among overlay nodes [7]. Per-node bandwidth consumption and storage requirement for such dissemination grow linearly with the increase of the overlay size, which can be problematic for large-scale services. In comparison, the Saxons structure management layer maintains periodically changing random membership subsets, which can satisfy many of the membership service needs with controllable bandwidth consumption and storage requirement.

In Saxons, each node maintains a dynamically changing *random-subset* data structure containing a number of other overlay members. The size limit of the random-subset (denoted by s) is determined according to the allowed per-node storage consumption. Membership subset queries are fulfilled through random selection from the local random-subset. Each node periodically disseminates a certain number of overlay members to each of its neighbors for updating the recipient's random-subset. The membership update size (denoted by k) and frequency are determined such that the bandwidth consumption is properly controlled. When random-subsets have reached their size limit s, randomly chosen old members are replaced by new members from membership updates.

The key for providing a random membership subset service with uniform representation over all overlay participants is to ensure such uniform representation in membership update sets. In our scheme, the update set from a node (called A) contains a selected portion of A's random-subset with uniform selection probability. A itself may also be included in each update set at probability k/n (n is the overlay size) to ensure its own equal representation. Since the overlay size n is often unknown at runtime, we use an approximate value $\tilde{n}_A = s/p$, where p is the proportion of all disseminated members received at A that are already in A's local random-subset at the time of receipt. Such an approximation is accurate when disseminated members received at A uniformly represent all overlay participants.

We provide simulation results to demonstrate the level of uniform randomness achieved by the Saxons random membership subset component. Simulations were run on a 6400-node randomly connected overlay structure with an average node degree of 8 and a degree upper-bound of 16. In the simulations, all the random-subsets are empty at the beginning of round 0. The length of a round is the average update interval between each pair of overlay neighbors. Each membership update contains 20 members in the simulations. Figure 3 illustrates the growth of the accumulated number of overlay members learned through membership updates (averaged over all 6400 nodes) since the overlay startup. Note that this metric is not equivalent to the number of members in each node's random-subset. It is a hypothetic metric whose growth over time illustrates the uniform randomness of membership updates. Results for several random-subset sizes are compared with the result for the uniformly random updates, which is produced when all membership updates contain uniformly random overlay nodes. The intuition for this comparison is that a membership update scheme that contains larger positive correlation among its update sets would generate slower growth in the number of learned overlay members.

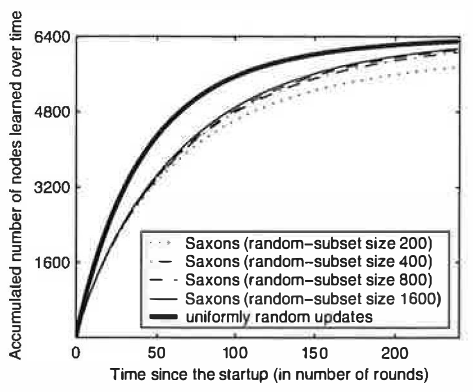


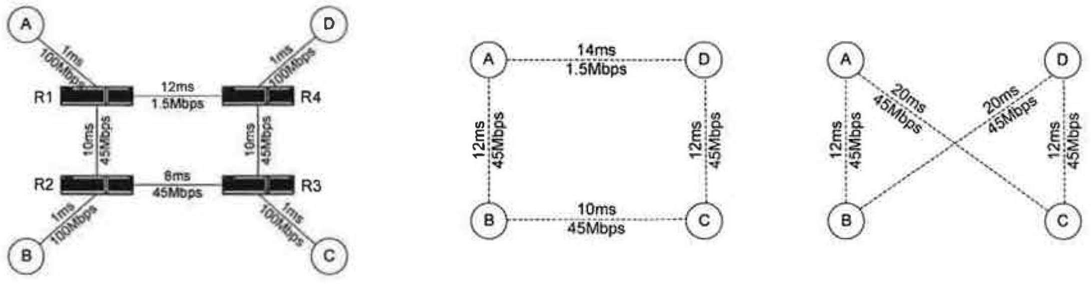
Figure 3: Growth of the accumulated number of overlay members learned from membership updates (averaged over all nodes) in a 6400-node overlay.

Results in Figure 3 show that the Saxons random membership subset component performs very close to the ideal uniform randomness. We also observe that the result is not very sensitive to the random-subset size, though larger random-subsets exhibit slightly higher uniform randomness. This is because each random-subset serves as a buffer to screen out non-uniformity in incoming membership updates and larger random-subsets are more effective on this.

3.2 Network Measurement

A fundamental problem for substrate-aware overlay service construction is how to acquire network latency and bandwidth data to support efficient overlay service construction. For latency measurement between two nodes, we simply let one node ping the other N_l times and measure the round-trip times. We remove the top 20% and bottom 20% of the measurement results and take the average of the remaining values. Pings could be conducted using ICMP ECHO messages or by employing a user-level measurement daemon responding to ping requests at each host.

Bandwidth measurement requires more consideration because it is harder to get stable results and it consumes much more network resources. Since the measurements would be conducted repeatedly in the runtime due to system dynamics, our goal is to acquire sufficiently accurate measurement data at a moderate overhead. The bandwidth measurement scheme we use is derived from the *packet bunch* technique proposed by Carter and Crovella [5] as well as Paxson [21]. Specifically, when node A wants to measure the bandwidth from node B, it sends out a UDP request to the measurement daemon at B, which replies back N_b UDP messages at the size of S_b each. A then records the receipt times of the first and



(a) Physical network (b) Overlay structure I (c) Overlay structure II

Figure 1: An example to illustrate overlay structure qualities.

Service types	Overlay latency	Hop-count distance	Overlay bandwidth
Short unicast	✓	✓$^{\times}$	×
Short multi/broadcast	✓	✓$^{\times}$	✓$^{\times}$
Long unicast	✓	✓$^{\times}$	✓
Long multi/broadcast	✓	✓$^{\times}$	✓
Neighbor comm.	×	✓	✓

Table 1: Importance of structure qualities for overlay services. A check mark "✓" means the metric is important to this type of services while a "×" represents the opposite. A mark "✓$^{\times}$" means the metric may be important under certain circumstances.

munications. Services with short-message communications generally do not place much demand on overlay bandwidth. However, short multicast/broadcast services like query flooding may incur high bandwidth consumption when a large number of queries flood the system simultaneously.

Table 1 summarizes these performance metrics and their importance to each of the five types of services we listed in Section 2.1. We believe that the above three lines of performance objectives cover the quality demand of a wide range of overlay services.

2.3 Other Design Objectives

In addition to constructing high quality overlay structures, we describe below several additional design objectives for Saxons. 1) **Scalability**: Recent measurements show that the main Gnutella network contains more than 100,000 nodes [16]. In order to support large-scale services, it may be infeasible to maintain the complete system view at any single node. 2) **Connectivity**: The fault or departure of a few bridging nodes may cause the partition of remaining overlay nodes, which could degrade or even paralyze the overlay service. Saxons provides partition detection and repair support such that upper-level services do not need to worry about the overlay partitioning. 3) **Stability**: Frequent node joins and leaves in a self-organizing system may cause instability in the overlay structure. Saxons is designed to hide such instability from upper-level services by maintaining the overlay structure quality with infrequent link adjustments. Structure stability is especially important for the performance of services that maintain link-related state, such as many routing services.

3 System Design

The Saxons overlay structure management layer contains six components. The bootstrap process determines how new nodes join the overlay structure. The structure quality maintenance component maintains a high quality overlay mesh while the connectivity support component actively detects and repairs overlay partitions. They run periodically to accommodate dynamic changes in the system. The above Saxons components are all supported by the membership management component that tracks a random subset of overlay members. The structure quality maintenance is further supported by two other components responsible for acquiring performance measurement data for overlay links and finding nearby overlay hosts. Figure 2 illustrates the six Saxons components and their relationship. Note that the connectivity support is an optional component in Saxons, which could be turned off for structures that are unlikely to be partitioned (*e.g.*, those configured with a high link density).

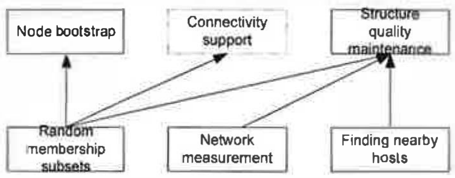


Figure 2: Saxons components.

The scalability of Saxons is achieved by controlling the system management overhead when the overlay scales up. The key guideline of our design is that the per-node management cost should only depend on the number of directly attached overlay links, not on the overlay size. In order to maintain a high level of robustness, Saxons employs a non-hierarchical or functionally symmetric architecture (with the exception of the optional Saxons connectivity support). Non-hierarchical designs are inherently free of scaling bottlenecks and they exhibit strong robustness in the face of random failures or even intentional attacks.

Our design and analysis in this paper assumes a fail-stop node departure model in which the departing node abruptly stops all actions at the exit time. We also assume there is no malicious participant in the overlay.

2 Design Objectives

Saxons is designed to provide efficient overlay connectivity support that can assist the construction of large-scale Internet overlay services. In order to assemble a comprehensive set of performance goals for the Saxons structure management layer, we first examine existing overlay services and categorize them based on similar communication patterns. Then we describe our objectives on overlay structure qualities and how they support our targeted services. We will also describe other Saxons design objectives.

2.1 Targeted Services

Below we list five categories of overlay communication patterns. We do not make claims on the completeness of this list but we believe it should cover the communication patterns of a large number of overlay services.

Short unicast: Services of this type involve single-source singe-destination short messages delivered along the overlay structure. An example of such services is the index-assisted object lookup [8], where the object lookup query messages are routed in the overlay network based on heuristics maintained at each node.

Short multicast/broadcast: Services of this type involve single-source multiple-destination short messages delivered along the overlay structure. An example of such services is the query flooding-style object lookup, such as Gnutella.

Long unicast: Services of this type involve a large amount of data delivered between two nodes in the overlay network. The large data volume makes it important to find efficient data delivery paths. Since the underlying Internet provides native unicast transport service, an overlay unicast delivery optimized for short latency only makes sense when the triangular inequality on Internet latencies does not hold, *e.g.*, due to the policy-influenced BGP routing or transient outages [1, 26].

Long multicast/broadcast: Due to the slow adoption of native multicast support in the Internet, there have been a large number of recently proposed overlay multicast protocols. In these protocols, large amount of streaming data is delivered between a single source and multiple destinations in the overlay network. These protocols focus on constructing multicast data distribution trees optimized for low latency [7], high bandwidth [14], or both.

Periodic pairwise neighbor communication: This communication pattern is frequently used for soft state maintenance in the overlay network, such as membership management [7], routing table maintenance, and Web proxy cache index maintenance. In the case of Web proxy caching, a large number of cooperative caches can maintain the information about each other's content through periodic pairwise neighbor communication in an overlay structure that connects them.

2.2 Objectives on Overlay Structure Qualities

The Saxons structure can be configured with different link density to offer tradeoff between structure simplicity and achievable overlay quality. Such a density is determined through a configurable node degree range in Saxons. Below we discuss performance objectives for the Saxons structure along three lines: the overlay latency, hop-count distance, and the overlay bandwidth.

The first performance objective is to achieve low *overlay path latency*, defined as the end-to-end latency along the shortest overlay path for each pair of nodes. The *relative delay penalty* (or *RDP*) is another common metric for overlay latency, which is defined as the ratio of the overlay path latency to the direct Internet latency. Consider a physical network illustrated in Figure 1(a). A, B, C, and D are edge nodes while R1, R2, R3, and R4 are internal routers. The latency and bandwidth for each physical link are indicated. Figures 1(b) and 1(c) illustrate two overlay connectivity structures both of which have a per-node degree of two. The latency and bandwidth of each virtual overlay link are also indicated with the assumption that the underlying substrate network employs shortest-path routing. Both connectivity structures provide the same overlay latency for A-B and C-D. Structure II allows slightly less latency for A-C and B-D while structure I provides much shorter paths for A-D and B-C. In average, Structure I is superior to Structure II in terms of the overlay path latency (15.33ms *vs*. 21.33ms) and RDP (1.03 *vs*. 1.58). The overlay latency is very important for the first four categories of services described in Section 2.1. For services with the periodic pairwise neighbor communication pattern, the data propagation delay between two nodes depends mostly on the overlay hop-count distance since the neighbor communication frequency is often independent of the link latency.

Our second objective is to achieve low *overlay hop-count distance*, defined as the hop-count distance along the overlay structure for each pair of nodes. The two structures illustrated in Figure 1 have the same average overlay hop-count distance of 1.33. As mentioned earlier, the hop-count distance in an overlay structure is important for services with the periodic pairwise neighbor communication pattern. We further argue that this metric also affects the performance of other services utilizing overlay structures. This is because an overlay route with larger hop-count may suffer more performance penalty in the presence of transient congestion or faults at intermediate nodes.

In terms of the overlay bandwidth, we are interested in two specific metrics: *bandwidth along the shortest overlay path* and *bandwidth along the widest path*. Bandwidth along the shortest overlay path is important when the path selection desires both low latency and high bandwidth. Bandwidth along the widest path is the essential metric when the upper-level service is predominantly interested in finding high-bandwidth overlay routes. For the two overlay structures illustrated in Figure 1, structure II provides high bandwidth (45Mbps) routes for all pairs of nodes while structure I has low bandwidth (1.5Mbps) along the shortest path for A-D. High overlay bandwidth is desired for long unicast/multicast/broadcast communications as well as periodic pairwise neighbor com-

Structure Management for Scalable Overlay Service Construction

Kai Shen
kshen@cs.rochester.edu
Department of Computer Science, University of Rochester

Abstract

This paper explores the model of providing a *common* overlay structure management layer to assist the construction of large-scale wide-area Internet services. To this end, we propose *Saxons*, a distributed software layer that dynamically maintains a selected set of overlay links for a group of nodes. Saxons maintains high-quality overlay structures with three performance objectives: low path latency, low hop-count distance, and high path bandwidth. Additionally, it provides partition repair support for the overlay structure. Saxons targets large self-organizing services with high scalability and stability requirements. Services can directly utilize the Saxons structure for overlay communication. Saxons can also benefit unicast or multicast overlay path selection services by providing them a small link selection base without hurting their performance potential.

Our simulations and experiments on 55 PlanetLab sites demonstrate Saxons's structure quality and the performance of Saxons-based service construction. In particular, a simple overlay multicast service built on Saxons provides near-loss-free data delivery to 4 times more multicast receivers compared with the same multicast service running on random overlay structures. This performance is close to that of direct Internet unicast without simultaneous traffic.

1 Introduction

Internet overlays are successfully bringing large-scale wide-area distributed services to the masses. A key factor to this success is that an overlay service can be quickly constructed and easily upgraded because it only requires engineering at Internet end hosts. However, overlay services may suffer poor performance when their designs ignore the topology and link properties of the substrate network. Various service-specific techniques have been proposed to adapt to Internet properties by selecting overlay routes with low latency or high bandwidth. Notable examples include the unicast overlay path selection [1, 26], measurement-based end-system multicast protocols [2, 7, 14], and recent efforts to add substrate-awareness into the scalable distributed hash table (DHT) protocols [6, 24, 32].

In this paper, we present the design and implementation of a distributed software layer providing *S*ubstrate-*A*ware Connectivity *S*upport for *O*verlay *N*etwork *S*ervices, or *Saxons*. Saxons constructs and maintains overlay connectivity structures with qualities such as low overlay latency, low hop-count distance, and high overlay bandwidth. Through a very simple API, service instances at overlay nodes can query the locally attached overlay links in the structure. Overlay services can directly use Saxons structure links for communication. They may also further select overlay links from the given structure based on application-specific quality requirements or service semantics.

Many popular overlay services, such as Gnutella, are *unstructured* in that they are designed to operate on any overlay network structures. Saxons works naturally with these services by providing them a high quality connectivity structure for overlay communication. Saxons can also benefit unicast [1] or multicast overlay path selection services [7] by providing them a small link selection base. These services would normally incur much higher overhead if they directly run on the completely connected overlay, which often limits their scalability [1, 7]. The Saxons overlay structure can be configured with different link density to offer tradeoff between overlay complexity and path redundancy or quality. A more connected Saxons structure would provide better *achievable* quality. However, it typically consumes more resources to find the optimal path in a structure with higher link density.

It should be noted that a general-purpose overlay structure layer cannot be easily integrated with strongly structured protocols where the protocol semantics dictates how overlay nodes should be connected. Prominent examples are the recently proposed scalable DHT protocols [23, 28]. However, we believe that the "strongly structured" nature of these protocols are not inherent to the DHT service itself. For instance, we have demonstrated that a scalable DHT service can be constructed on pre-structured overlay networks [27].

We should also emphasize that the primary goal of this work is to provide a general easy-to-use software layer with wide applicability. Achieving optimal performance for individual services is not our focus. The rest of this paper is organized as follows. Section 2 describes our objectives and Section 3 presents the system design in detail. Section 4 illustrates our simulation-based evaluation results. Section 5 and 6 describe a prototype Saxons implementation and service constructions on the PlanetLab testbed. Section 7 discusses related work and Section 8 concludes the paper.

References

[1] Mark B. Abbott and Larry L. Peterson. A Language-Based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[3] David P. Anderson and Lawrence H. Landweber. A grammar-based methodology for protocol specification and implementation. In *Proceedings of the ninth symposium on Data communications*, pages 63–70. ACM Press, 1985.

[4] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, pages 165–175, 2002.

[5] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. In *Proceedings of the First International Symposium on Formal Methods for Comp onents and Objects*, Leiden, The Netherlands, March 2003.

[6] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[7] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–333, May/June 1999.

[8] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[10] Frank Dabek, Ben Zhao, Peter Drushcel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *2nd International Workshop on Peer-to-peer Systems (IPTPS'03)*, February 2003.

[11] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[12] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, August 2001.

[13] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

[14] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM*, pages 3–13, 1999.

[15] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[16] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[17] Massachusetts Institute of Technology. *lsd*, 2004. http://-www.pdos.lcs.mit.edu/chord/.

[18] Fabric Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Imp lementation (OSDI'2000)*, San Diego, California, October 2000.

[19] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[20] Rice University. *FreePastry*, 2004. http://www.cs.rice.edu/-CS/Systems/Pastry/FreePastry/.

[21] Adolfo Rodriguez, Dejan Kostić, and Amin Vahdat. Scalability in Adaptive Multi-Metric Overlays. In *The 24th International Conference on Distributed Computing Systems (ICDCS)*, March 2004.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[23] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[24] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *Third International Workshop on Networked Group Communication*, November 2001.

[25] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[26] Peter Urban, Xavier Defago, and Andre Schiper. Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In *ICOIN*, pages 503–511, 2001.

[27] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[28] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsu n Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *Fourth International Conference on Verification, Model Checking and Abstrac t Interpretation*, pages 283–297, New York, January 2003.

[29] Ellen W. Zegura, Kenneth Calvert, and M. Jeff Donahoo. A Quantitative Comparison of Graph-Based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[30] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[31] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.

Such scale limitations are overcome by network emulation such as with ModelNet [27]. It enables the emulation of native IP applications by subjecting packets to link restrictions as specified by a network topology. It emulates routers' queuing delay and congestion. In our experiences, thousands of overlay nodes can run on 20-50 commodity PCs in the ModelNet environment. The same code runs unmodified in production Internet environments and testbeds, including PlanetLab [19]. As a result, ModelNet's accuracy and scalability makes it an appropriate choice for large-scale evaluation.

Neko [26] is another environment for developing and evaluating distributed algorithms. Similar to MACEDON, it allows the same algorithm specification to be used in a simulator and in a live system. MACEDON, however, provides a DSL, a set of libraries that address common issues in distributed algorithm development, and a generic API that facilitates interoperability between overlay algorithms and applications.

5.2 Related Languages

MACEDON is broadly related to domain-specific languages (DSLs) that typically generate functional code from domain-specific representations. Teapot [7] is a DSL for writing cache-coherence protocols. Like MACEDON, Teapot describes protocol behavior with the use of event-driven finite state machines. Teapot can generate "continuations" that allow nodes to suspend processing while waiting for a particular event. Unlike MACEDON, code generated by Teapot is not self-contained since the user must hand-code message handling functions. Additionally, Teapot's target domain (cache coherence protocols) is somewhat smaller than MACEDON's domain. Another domain-specific language is the Devil Interface Description Language (IDL) [18] designed for a substantially different domain than MACEDON, but is related in design. IDL can be used as documentation for hardware interfaces and can help driver development by reducing the burden of low-level programming. Devil also includes semantics for verifying specifications.

There has been substantial research in network protocol specification and implementation. RTAG [3] uses a context-free attribute grammar for protocol specification, emphasizing simplicity and portability. The grammar is used to capture event sequences allowed by the protocol. Morpheus [1] is an object-oriented language tailored for high-performance protocol implementations. It constrains a protocol designer to a set of design disciplines derived from experience, advocates the use of simple protocols that are selected and combined at runtime, and capitalizes on the knowledge of common patterns in protocol processing to optimize generated object code. Prolac [14], a lightweight object-oriented language, focuses on readability, modularity and extensibility. Its authors offer positive experiences with a TCP implementation. Prolac's *actions* allow arbitrary C code to be included; it is inserted into the C code produced by the Prolac compiler. Relative to these efforts, MACEDON is specifically geared toward overlay networks, focusing on a standard API, explicit support for protocol layering, and language support for common overlay functionality.

Beyond system specification, a number of languages target high-level design and protocol verification. These range from the highly mathematical, such as IOA [28] to more programmatic languages, such as TLA [5]. In contrast to MACEDON, neither generates functional code. IOA is an Input/Output Automaton specification language, allowing designers to specify one or more automatons to describe their system. IOA tools perform simulated execution that suggest likely invariants and automatically prove seemingly tedious portions of system specification. for formal verification. TLA is a high-level specification in a highly mathematical language. It is intended to be a design aid, and, combined with its model checker, can be used to find and remove flaws from system designs before system implementation.

6 Conclusions and Future Work

We have presented MACEDON to facilitate the design and implementation of overlay algorithms. Our system provides a domain-specific language for specifying the high level behavior of overlays such as DHTs and application-level multicast. MACEDON provides a common infrastructure that enables fair and consistent overlay evaluation. We make use of an overlay-generic API that enables protocol layering and facilitates porting applications from one overlay to another. Our results show that MACEDON can greatly decrease development and evaluation effort while yielding overlay implementations that closely resemble or outperform published results, including those for AMMO, Bullet, Overcast, NICE, Chord, Pastry, Scribe, and SplitStream. We believe that MACEDON can be used as an educational tool to understand the intricacies of overlay algorithms. Finally, we believe that the MACEDON vision extends beyond overlay algorithms to include a wider class of distributed algorithms, though this is the subject of future work.

Acknowledgments

We would like to thank David Becker and Ken Yocum for their help with ModelNet. In addition, we thank our shepherd, Timothy Roscoe, Rebecca Braynard, and anonymous reviewers who provided excellent feedback.

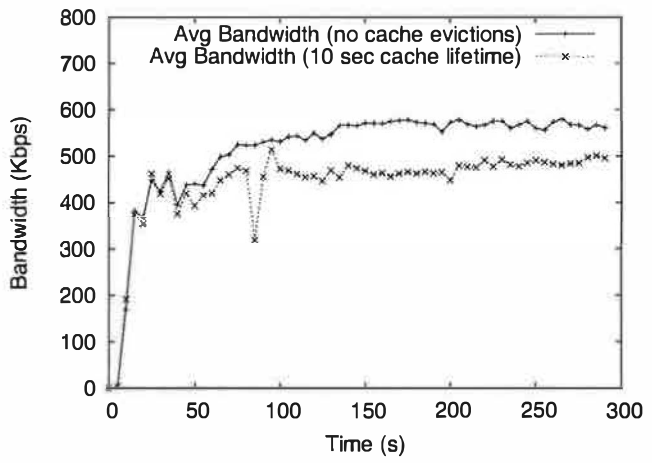


Figure 12: SplitStream bandwidth for two cache policies.

Disparate evaluation techniques have led to the use of many different performance evaluation metrics in overlay comparison. While an evaluation may be concerned with low link stress, it is unclear whether it is relevant to all applications. A high-bandwidth link could have link stress of hundreds for a chat-like application and perform well, while a link stress of two over a modem link for a video distribution would likely be unacceptable. As a result, it is challenging to choose the appropriate evaluation metric. MACEDON attempts to bridge this gap by providing a framework that can report a variety of popular evaluation metrics. We believe that MACEDON will encourage evaluation across more performance metrics, allowing the metrics themselves to be evaluated.

Our SplitStream experiments are designed to demonstrate MACEDON's ability to experiment with a variety of protocol features. For these tests, we created 300-node SplitStream forests. We developed a multicast application that streams 1000-byte packets at a predetermined rate (600Kbps for our experiments). Only one node is designated as the stream source while all other nodes join the multicast session as receivers. We first allow Pastry routing tables to converge by idling the system over the 300 seconds. Figure 12 shows the resulting per-node average bandwidth over time after the convergence period for two SplitStream flavors. SplitStream and Scribe use `macedon_routeIP()`, requesting that data be delivered directly over IP. Pastry does this by maintaining a *location cache* that maps hash addresses to IPs. Cache entries have an associated lifetime, thereby avoiding stale mappings that could lead to inefficient routing (a node could receive packets for a hash address it no longer owns). With cache eviction disabled, SplitStream delivers an average of 580 Kbps to each node (since no additional nodes enter the overlay, cache entries remain correct). With a one-second cache lifetime, bandwidth drops to 500 Kbps as additional bandwidth is consumed to re-establish stale cache entries. In summary, we believe that MACEDON is appropriate for carrying out such detailed and uniform protocol comparisons.

5 Related Work

MACEDON currently supports two types of overlays, distributed hash tables (DHTs) [11, 22, 25, 30] and application level multicast [4, 6, 12, 13, 15, 16, 22]. DHTs and their applications [9, 24, 23, 31] use hashing to map data objects and nodes to a logical address space for request routing. The hash value of a node determines which portion of the hash address space it *owns* and therefore, which data objects it will serve. By ensuring sublinear node (routing table) state and overlay width and depending on uniform server distribution (using consistent hashing), these overlay algorithms exhibit high performance and scalability.

Built on top of Pastry (or any other DHT), Scribe [24] creates multicast distribution trees rooted at the DHT node owning the multicast session ID. Receivers enter the session by routing join requests toward the root. Intermediate nodes along the path subsequently create a reverse path forwarding tree. Building on Scribe's success, SplitStream [6] uses multiple Scribe trees for data striping, thereby achieving higher bandwidth.

Other popular multicast overlays do not make use of DHTs. Most, including Overcast [13], NICE [4], and AMMO [21] create distribution trees optimized toward application-specific performance. In contrast, Bullet [16] creates a mesh where nodes exchange *summary tickets* that are used to select data peers. Nodes with disjoint data peer with one another. Since data is received from a number of carefully selected peers, Bullet nodes receive much higher bandwidth relative to tree-based overlays.

5.1 Evaluation Methodologies

The ns [29] network simulator provides a standard framework for accurate simulation of network protocols. Unfortunately, packet-level, congestion-aware simulation is costly, leading to inadequate scaling properties when evaluating overlays over a few hundred in size. For smaller-scale scenarios, ns provides an efficient and inexpensive mechanism for system evaluation. In the end, many researchers have created their own simulators, sacrificing accuracy for scale. These simulators tend to provide packet-level simulation but fail to account for congestion, packet loss, and queuing delays.

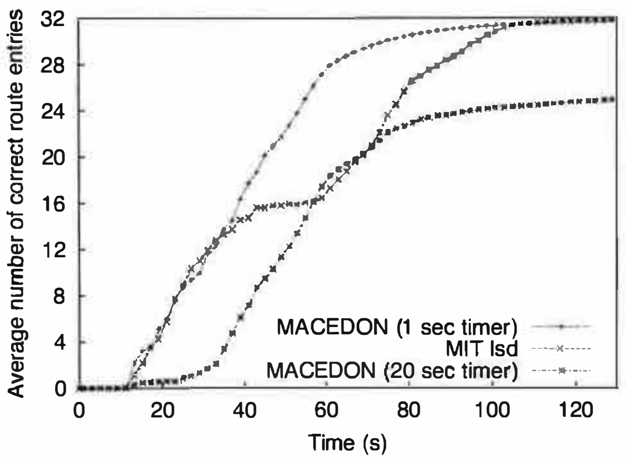


Figure 10: Convergence toward correct routing tables for MIT and MACEDON Chord implementations.

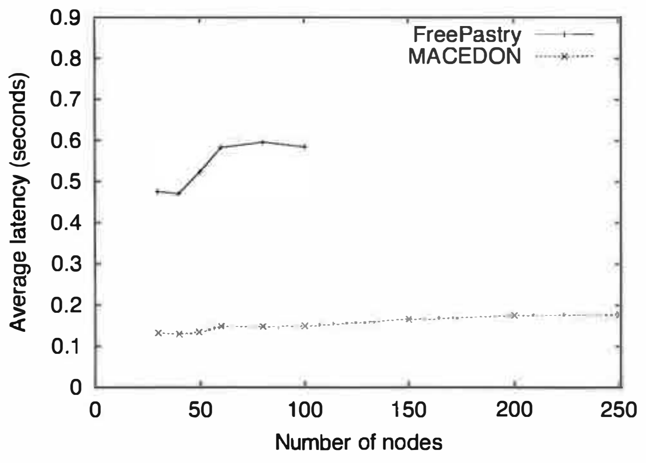


Figure 11: Average latency of received Pastry packets.

4.2.3 Pastry

One goal of the MACEDON framework is to enable rapid prototyping of distributed systems while maintaining the performance and low-level optimizations available from hand-crafted C/C++ implementations. As one initial validation of our success against this metric, we compare the performance of the Pastry algorithm [22] implemented in MACEDON and within FreePastry [20]. MACEDON provides a high-level specification language with many of the same benefits of Java, along with libraries and routines specifically tailored to DHTs and overlays. However, it produces C++ code that does not suffer from some of the memory and performance overheads associated with Java and RMI. Our MACEDON Pastry implementation consists of 400 semicolons versus approximately 1,500 semicolons in the Java FreePastry implementation[1]. To quantify these benefits, we developed a simple test application to validate our Pastry implementation. Each application instance streams at some target data rate (10Kbps in this example) by sending 1000-byte packets at the given interval. On each data send, the application chooses a destination ID uniformly at random from the hash address space.

We estimate end-to-end delays for MACEDON Pastry and FreePastry [20] (using the RMI protocol). We varied the number of randomly selected nodes from our 20,000-node topology. For both systems, we allowed routing tables to converge for 300 seconds before streaming data. Due to our low streaming rate, intended targets received essentially all packets transmitted to them. For both systems, each node received approximately the same number of packets corresponding to the size of hash address space portion it owns. Figure 11 illustrates the average per-packet delays. We were unable to run FreePastry beyond 100 participants (two instances per physical machine) due to insufficient memory on our hardware. We have successfully run 20 MACEDON instances on these same machines. The graph shows that average latency in MACEDON is approximately 80% lower than in FreePastry, largely attributable to Java's RMI overhead. While FreePastry's "wire" protocol has yielded more favorable results (comparable to MACEDON in some cases), it is unstable in the current FreePastry release. Overall, our results show promise for MACEDON's ability to enable rapid prototyping while maintaining system performance.

[1]The FreePastry distribution consists of almost 15,000 semicolons, with significant functionality beyond Pastry. Our estimate is a conservative count based on manual inspection of the source tree.

4.3 Comparing Overlays

One important contribution of MACEDON is the creation of a fair and consistent overlay evaluation framework. To this end, MACEDON generates native TCP/IP code, allowing it to leverage ModelNet emulation and live deployment across the Internet, including the PlanetLab testbed (to support simulation environments, we also provide limited ns compatibility). MACEDON can automatically extract vital topology information from ns and ModelNet, allowing it to evaluate overlays against a wide array of metrics. Without such global information, it is impossible to accurately gauge an overlay's performance under certain metrics. For example, MACEDON can extract routing tables from ns and ModelNet to report the expected performance along metrics such as link stress, latency stretch, and relative delay penalty (RDP).

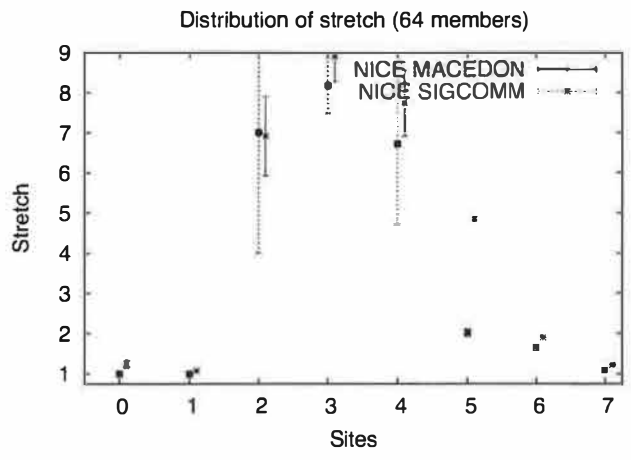


Figure 8: Observed stretch for published and MACEDON NICE implementations.

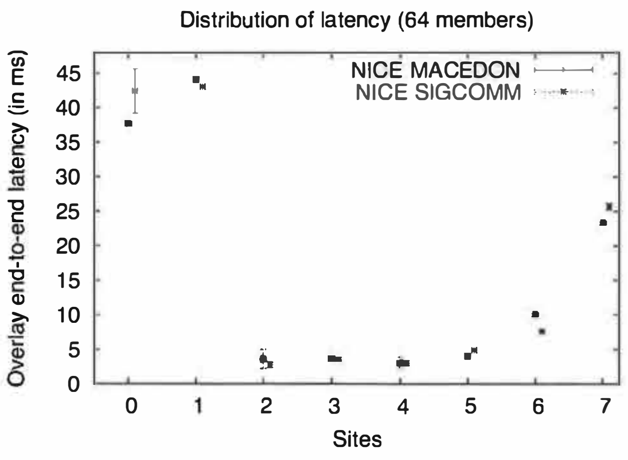


Figure 9: Observed latencies for published and MACEDON NICE implementations.

allows us to extract features of the resulting overlay by using ModelNet routing and topology information. For all other experiments, we use 20,000-node INET [8] topologies with varying numbers of clients (200–1000). In all cases, we run our experiments on (2–50) 1.4Ghz Pentium-III machines running Linux 2.4.23. We multiplex multiple node instances on these machines. All traffic passes through our 1Ghz Pentium-III ModelNet cores running FreeBSD-4.9. While all results in this paper use ModelNet, we note that we have successfully run smaller experiments (50+ nodes) over PlanetLab [19] (refer to [16] for sample results).

4.2.1 NICE

To validate our implementation of NICE, we run the same experiments described in [4] for small-scale Internet scenarios (64 nodes) and compare our results with the published values. Figures 8 and 9 show the average observed stretch and latency for NICE nodes in each of eight different Internet sites as reported in Figures 15 and 16 in [4] versus our MACEDON implementation. We slightly offset the MACEDON values to the right for clarity. Our results closely match the published results, with only a minor discrepancy in one of the sites. We believe this is due to our implementation lacking the probe time binning strategy presented in [4], though adding this to our implementation is straightforward.

4.2.2 Chord

We validated our MACEDON Chord implementation by comparing it to the MIT distribution, lsd. We used a 20,000-node INET topology with 1000 Chord participants for this experiment. We made two modifications to the MIT code to first dump all routing tables every two seconds (something already available in the MACEDON implementation via debugging features) and to use a smaller hash function, since our implementation of Chord only uses a 32-bit hash address space (nodes hash to the same hash address in MACEDON and lsd). We calculated correct routing tables for each node given global knowledge of all nodes joining the system.

Figure 10 shows the convergence of routing tables toward the correct values over time (averaged per-node) for MACEDON and lsd. The graphs shows two MACEDON curves, corresponding to two different settings of the "fix fingers" timer. This timer triggers Chord to route a repair request message to a random finger (routing) table entry. The ultimate destination of this message responds, allowing the requesting node to verify the correctness of that route entry. While the lsd code dynamically adjusts the period of the fix fingers timer, our current MACEDON implementation only supports static periods (1 and 20 seconds in this experiment).

The optimal strategy for dynamically adjusting protocol parameters such as timer periods is unclear. For example, our static 1-second strategy outperforms lsd's dynamic strategy. The converse is true with a 20-second timer setting, as convergence is much slower in this case. In both MACEDON cases, routing tables converge steadily as nodes join the Chord ring, eventually leveling off once all nodes have joined. In lsd, convergence is not as steady as fix fingers timers are dynamically adjusted. The goal of this experiment is to demonstrate MACEDON's ability to match or exceed that of lsd. Further, we note that MACEDON enables researchers to more effectively explore different dynamic timer strategies.

bor list. Line 9 adds a neighbor while line 7 shows how to clear a neighbor list. Neighbor lists can be queried as in line 14 ("from" is the source address of the inbound "join_reply" message) and accessed directly as in line 16. Finally, lines 5 and 13 illustrate selecting a random entry from a neighbor list.

Typically, overlays compare potential edges along some performance metric, such as round-trip time (e.g., NICE). Overcast estimates bandwidth by measuring the delay associated with receiving some number of probes at a sustained bandwidth. Line 17 shows how neighbor entries store this information. Additional neighbor entry fields could be maintained in such a manner.

3.3.3 Explicit Thread Serialization

While locking behavior is specified on transition declarations, an overlay developer may required explicit access to an agent's (protocol instance) lock. That is, conditions under which locking is required may depend on intricate behavior within the transition itself. In this case, the transition could employ the `Lock_Write()`, `Lock_Read()`, and `Unlock()` primitives. In our experience, however, transition-based locking has been adequate for all the overlays we have considered.

4 Evaluation

In this section, we evaluate MACEDON's ability to: i) facilitate overlay design, implementation, and evaluation, ii) implement a broad range of algorithms with good performance and scalability characteristics, and iii) enable comparisons of competing overlay technologies. While it is not practical to prove that MACEDON will be able to meet the demands of all distributed algorithms, we use our success with of a broad variety of modern overlays to support our goal of qualitatively improving the way overlay research is conducted.

4.1 Expressiveness

One key contribution of this work is the implementation and validation of a broad range of network overlays in the MACEDON environment, including: AMMO [21], Bullet [16], Chord [25], NICE [4], Overcast [13], Pastry [22], Scribe [24], and SplitStream [6]. Figure 7 summarizes the lines of code (LOC) counts for each of these MACEDON specifications. NICE, being a more complex protocol than all others required approximately four weeks of skilled programmer's time to implement and debug. Its MACEDON specification is approximately 500 LOC

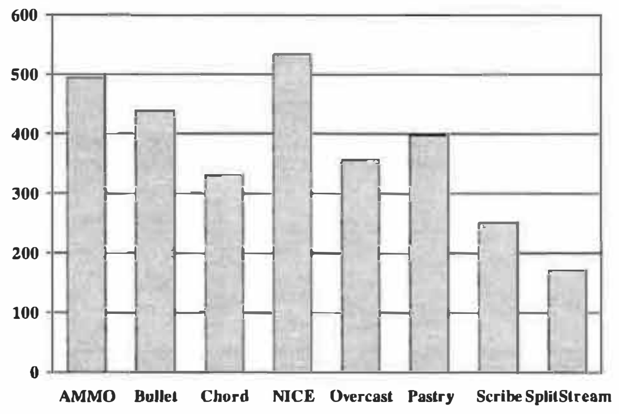


Figure 7: Lines of code used in various algorithm specifications.

while its generated C++ code is over 2500. The MACEDON operating environment is around 3500 LOC, yielding an estimated total of 6000 C++ LOC to completely implement NICE from scratch.

On the other end of the spectrum, SplitStream's MACEDON specification is under 200 lines of code, primarily because SplitStream, being layered on top of Scribe and Pastry, exploits functionality provided by those systems. Implementing SplitStream also required small changes to our Scribe implementation, primarily since the description of SplitStream [6] requires changes to Scribe's "pushdown" function. Though SplitStream and Scribe are originally designed to run over Pastry, we note that MACEDON's layering feature in conjunction with its standard API allows us to switch underlying DHT layers easily. For instance, while our experiments show results for SplitStream running over Pastry, we are currently experimenting with using Chord as the underlying DHT.

4.2 Validation

This section provides validation of a subset (abbreviated for space reasons) of our MACEDON-generated implementations as compared to published results or freely available code distributions (MIT's lsd Chord [17] and FreePastry [20]). We further note that results included in [16] and [21] were achieved through MACEDON using the mac specifications described in this paper.

We believe that our results confirm the generality, accuracy, and performance of our infrastructure. We use the ModelNet [27] infrastructure to emulate large-scale Internet topologies, capturing hop-by-hop congestion and queuing behavior. For our NICE validation, we used extracted information from [4] to re-create the authors' Internet-like topology. Our evaluation infrastructure

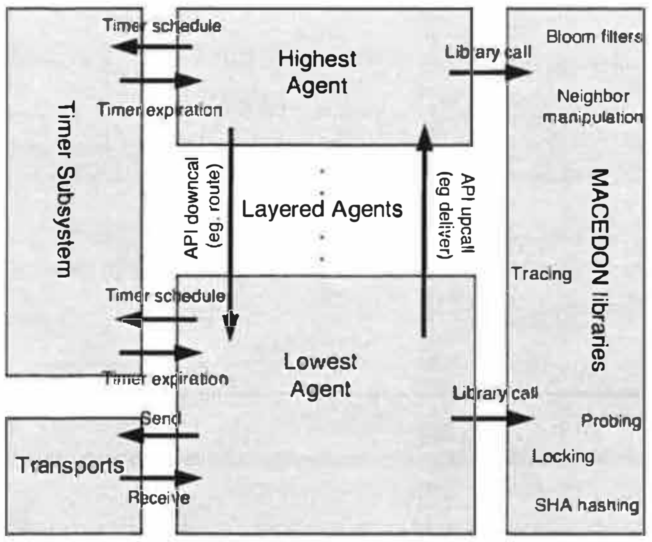


Figure 5: MACEDON agents.

tiplexing function calls the appropriate transition function based on the node's current state and the message type. MACEDON translates API specifications and timer transitions in much the same way.

Figure 5 outlines the resulting structure of MACEDON agents. MACEDON subsystems are implemented with thread pools that process timer and transport events. Along with application threads, they invoke transitions in agents. Timers can be employed by any layer in the MACEDON stack. However, only the lowest-layer agent may interact directly with the transport subsystem. Likewise, only the highest-layer agent interacts directly with the application. Though this example only shows two layered agents, MACEDON supports layering an arbitrary number of agents.

3.3 Specifying Actions

This section describes how an overlay developer invokes transition actions in MACEDON. While this could be done solely in the target programming language (i.e. C++), MACEDON provides libraries for invoking commonly-used actions, including the necessary functions to interface with our timer and transport subsystems as well as invoking cross-layer upcalls and downcalls. The MACEDON library collection is extensible, allowing users to add their own library routines. For example, we have created a library that manipulates bloom filters. The remainder of this section describes support for other commonly-used actions based on a sample transition of the Overcast specification, given in Figure 6. Line 2 of our sample transition shows how to access the "response" field of the incoming "join_reply" message.

```
 1 joining recv join_reply {
 2   if (field(response) == 1) {
 3     if (neighbor_size(papa)) {
 4       neighbor_oparent *pops =
 5                neighbor_random(papa);
 6       route_remove(pops->ipaddr, 0, 0, -1);
 7       neighbor_clear(papa);
 8     }
 9     neighbor_add(papa, from);
10     state_change(joined);
11     timer_resched(probe_requester, PINT);
12     neighbor_oparent *pops =
13               neighbor_random(papa);
14     if (neighbor_query(brothers, from)) {
15       neighbor_ochildren *newp =
16         neighbor_entry(brothers, from);
17       pops->delay = newp->delay;
18     }
19    upcall_notify(papa, NBR_TYPE_PARENT);
20   }
21   else {
22     if (neighbor_size(papa)) {
23       state_change(joined);
24     }
25     else { ... // omitted }
26   }
27 }
```

Figure 6: A sample Overcast transition.

Perhaps the most basic action specified within transitions is in changing system state, as specified in lines 10 and 23. Line 11 shows how we invoke the MACEDON timer subsystem to schedule a timer event. Finally, we demonstrate an upcall invocation in line 19.

3.3.1 Transmitting Messages

Overlay protocols transmit messages via lower layers (or underlying network substrate). MACEDON has built-in transmission primitives of the form:
⟨API⟩_⟨msg⟩(⟨dest⟩, ⟨fields⟩, ⟨buffaddr⟩, ⟨buffsize⟩, ⟨pri⟩)
Line 6 of our sample Overcast transition illustrates how we transmit the remove message to our "old" parent once we have determined that a move will occur. By specifying a buffer address and size of zero, this message will not be appending application data. Finally, the -1 priority requests use of the message's default transport.

3.3.2 Neighbor Management

MACEDON provides primitives to simplify neighbor list management. Our sample transition makes heavy use of these facilities. Lines 3 and 22 illustrate the `neighbor_size` function that returns the size of a neighlanguage types and neighbor sets. Communication in MACEDON can be reliable, congestion-friendly (using TCP), unreliable, congestion-unfriendly (using UDP), or reliable, congestion-unfriendly (using a simple sliding window protocol, SWP). It is sometimes advantageous to use *multiple* blocking transports (e.g. TCP) of the same type. This is particularly evident when one message has higher priority than another. If the transport is blocked sending low priority messages, it is unable to send any available high priority messages until the connection is unblocked. By defining multiple transport based on priority, this problem is easily overcome. For example, Overcast includes:

```
transports {
  SWP HIGHEST;
  TCP HIGH;
  TCP MED;
  TCP LOW;
  UDP BEST_EFFORT;
}
messages {
  BEST_EFFORT join { }
  HIGHEST join_reply { int response; }
  HIGHEST probe_request { ... // fields omitted }
}
```

Overcast includes three TCP transports, as well as one SWP and one UDP, and associates each message to the appropriate transport. In higher layers, a specification associates messages with a default *service class* or priority. A higher layer invokes the layer below to transmit the message, passing the desired priority along with it. The lower layer determines how to process the message at the given priority.

The TRANSITIONS section describes the bulk of an overlay's behavior. The developer uses a set of MACEDON primitives to describe the actions that result from triggered events. All transitions are scoped by a FSM state expression, thereby allowing a protocol to specify different behavior based on its current system state. A developer may specify transition-specific options, such as write versus read serialization (write semantics are assumed by default). There are three types of transitions: API, timer, and message. While our Overcast specification is too large to include here, we summarize a few transitions (with actions removed for brevity):

```
transitions {
  any API route [locking read;] { ... }
  probing timer keep_probing [locking read;] { ... }
  !(joining|init) recv join { ... }
}
```

An API transition enables layers to communicate with layers directly above and below in the MACEDON stack. The "init" API is called by a higher layer to initialize protocol state and schedule necessary timers. "route", "routeIP", "multicast", "anycast", and "collect" represent requests to transmit data. Our example shows the declaration of Overcast's route API with read locking semantics. "create_group", "join", and "leave" are control calls for managing multicast session state. The remaining API calls represent atypical or extensible calls into the code, including notifying upper layers of a changed neighbor set, generic "upcall_ext" and "downcall_ext" to provide extensible specification of layer-to-layer collaboration, and failure detection ("error"). Our current implementation assumes the failure of a peer node if no message has been received from it in f seconds, a configurable parameter. If communication has ceased for $g < f$ seconds (another parameter), MACEDON triggers a heartbeat request/response sequence to solicit communication. Appropriate failure detection is an ongoing area of research. We consider MACEDON to be an appropriate framework for such research.

A timer transition occurs upon a timer expiration. In Overcast, the "keep_probing" timer fires when a node is transmitting probes. In this case, the node is in the "probing" state and follows read locking semantics since no node state is modified within the transition. Finally, a message transition is called in response to message reception. In addition to state scoping, these transitions are scoped by message type, enabling different transitions for different messages. In MACEDON, messages are *delivered* (this is the final destination) or *forwarded* (this node should forward the message). In our example, we have specified a join message reception when the state matches the expression "`!(joining|init)`", i.e. the Overcast node is in "joined", "probing", or "probed" states. This transition modifies state variables and makes use of the default write locking semantics.

3.2 Code Generation

MACEDON generates API-consistent code, termed the MACEDON *agent*, from an algorithm's specification. MACEDON parses a specification and translates it into executable C++ code that uses library functions and the MACEDON code engine including timer and transport subsystems (we also have partial support for generating ns code for better reproducibility of results). The engine and code libraries are common to all overlay implementations, increasing evaluation consistency and code reuse. While our current infrastructure does not yet support other programming languages such as Java, it is the subject of ongoing work.

The translation phase involves the declaration of protocol messages, states, neighbor types, state variables, and transitions. We create a demultiplexing function to receive data packets from a MACEDON interoperability layer that in turn interfaces with ns or native TCP/IP sockets. Upon receiving a message, the demul-

```
<PROTOCOL SPECIFICATION>: <HDRS>
  <STATE AND DATA><TRANSITIONS><ROUTINES>(0,1)
<HDRS>: "protocol" <name> ["uses" <base>](0,1)
  "addressing " ["hash"|"ip"]
  "trace_" ["off"|"low"|"med"|"high"]
<STATE AND DATA>: <CONSTANTS> <STATES>
  <NEIGHBOR TYPES> <TRANSPORTS>(0,1)
  <MESSAGES> <STATE VARS>
<STATES>: "states { " [<name> ";"]* "}"
<TRANSPORTS>: "transports {" <TRANSPORT>+ "}"
<TRANSPORT>: ["TCP"|"UDP"|"SWP"] <name> ";"
<MESSAGES>: "messages {" <MESSAGE>* "}"
<MESSAGE>: <transport name>(0,1) <name>
  "{" <MESSAGE FIELDS>* "}"
<STATE VARS>: "auxiliary data {"
  [<LOCAL VAR>|<NEIGHBOR VAR|<TIMER VAR>]*"}"
<NEIGHBOR VAR>: "fail_detect"(0,1) <name>
  <size>(0,1) <size>(0,1) ";"
<TIMER VAR>: "timer" <name><period>(0,1) ";"
<TRANSITIONS>: "transitions {"[<STATE EXPR>
  [<API TRANS>|<TIMER TRANS>|<MESSAGE TRANS>]
  <TRANS OPTIONS> "{" <code> "}"]* "}"
<API TRANS>: "API " <API TYPE>
<API TYPE>:  "init"|"route"|"routeIP"|
  "multicast"| ... |"join"|"upcall_ext"
<TIMER TRANS>:  "timer" <name>
<MESSAGE TRANS>: ["forward"|"recv"] <message>
```

Figure 4: MACEDON grammar highlights.

macedon_leave(), specifying the group value. Similar to macedon_route(), macedon_multicast() requires a session's ID instead of a node's destination address. macedon_collect() introduces a new primitive to traditional overlay APIs. It essentially performs the opposite of multicast, where data originates at non-root nodes and is collected via the distribution tree toward the root. Intermediate nodes can summarize data in an application-specific manner, ultimately delivering a global summary to the tree's root. We believe that a number of applications could benefit from this communication paradigm.

3 MACEDON Framework

This section describes how a developer can specify overlay behavior in MACEDON. We give an overview of the language and discuss its expressiveness. We also describe how MACEDON captures subtle implementation details that greatly influence overlay performance.

3.1 Grammar Overview

Figure 4 highlights the MACEDON language grammar. It allows a developer to define a PROTOCOL SPECIFICATION that MACEDON translates into working code. There are three main headers in mac file. The protocol header specifies the name of the protocol and optionally a base protocol for layering. For example, one could specify "protocol scribe uses pastry" to run Scribe over Pastry, or "protocol scribe uses chord" to change the underlying DHT. In this manner, one could perform a direct comparison between the two DHTs in support of application-layer multicast. The addressing header specifies whether the protocol uses IP- or hash-based addressing. One could add other types of addressing, for example, to test new hashing algorithms or node identifier schemes. The tracing header can be set to any of four increasing levels of automatic tracing.

The STATE AND DATA section includes definitions of states, neighbor types, transports, messages, and state variables. The STATES portion defines the allowed set of protocol FSM states. The "init" state is automatically generated as the starting state for all protocols. For Overcast, state definitions are: (refer to Figure 1):

```
states { joining; probing; probed; joined; }
```

The NEIGHBOR TYPES section specifies the sets of neighbors the protocol tracks (and their maximum number). Neighbor types may specify optional fields, such as delay, to track on a per-neighbor basis. Note that a field might itself be a set of neighbors. Returning to our example, Overcast nodes have parent and children neighbors:

```
neighbor_types {
  oparent 1 { ... // fields omitted }
  ochildren MAX_CHILDREN { ... // fields omitted }
}
```

The protocol also specifies persistent state variables. In addition to standard language types, neighbor sets, and multidimensional arrays of such types, state variables can specify timers with a specified expiration period. A neighbor list may be labeled "fail_detect", instructing MACEDON to monitor these neighbors for failure. Upon detecting failure, MACEDON will invoke an overlay's error API transition. Our Overcast specification includes the following state variables:

```
state_variables {
  oparent papa;  // parent neighbor
  ochildren kids;  // children neighbors
  oparent grandpa;  // grandparent neighbor
  ochildren brothers;  // sibling neighbors
  int probed_node;  // node we are probing
  int probes_to_send;  // count of probes left
  timer keep_probing;  // timer Z
  timer probe_requester;  // timer Q
  ...  // fields omitted for brevity
}
```

In MACEDON, the lowest-layer protocol specifies the transports it uses and associates transport instances with each message via TRANSPORTS and MESSAGES definitions. Messages may contain many fields, including standard

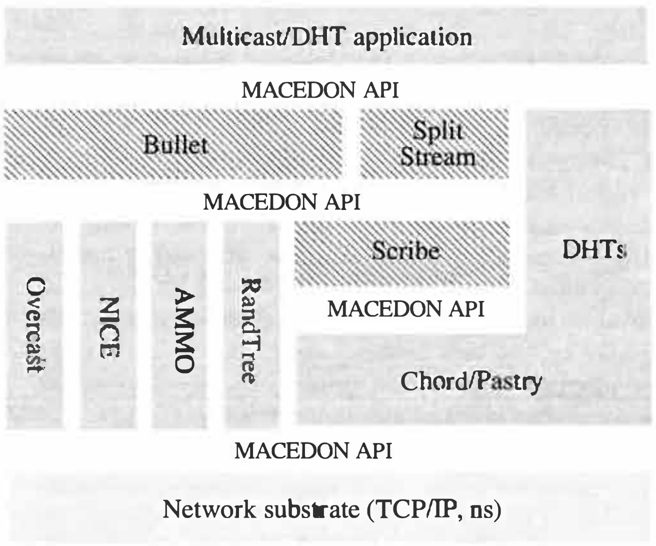


Figure 2: The MACEDON protocol stack.

data through the overlay. While sometimes obfuscated in design, we believe it is imperative for overlay implementations to provide appropriate APIs to application developers. A number of recent efforts [10, 22] have made initial steps at creating a single, standard API. We adopt an API similar to [10] and further enable API extensibility for protocol-specific functionality.

A standard API enables MACEDON applications to select underlying overlays without modification. In general, overlays support multicast or route primitives that route data from a source to destination(s) *through* the overlay. Typically, overlays provide upcalls at each routing hop so that intermediate nodes can perform application-specific functionality. For example, an intermediate Scribe node receiving a join request for a group will add the group to its list of multicast sessions and propagate the request toward the destination, thus building a reverse-path distribution tree.

Protocol layering (Figure 2) is central to implementing algorithms in MACEDON. The MACEDON protocol stack is divided into three components: application, multiple protocol layers, and network substrate (ns or TCP/IP). Much like the TCP/IP stack, higher layers in MACEDON use the services of lower layers. Bullet, for example, uses a simple randomly constructed tree, RandTree, for baseline data distribution.

Figure 3 illustrates a simplified version of the API that MACEDON overlays export. We provide an extensible upcall and downcall mechanism to perform protocol-specific collaboration across layers in the stack. As instances of this mechanism, we describe `forward()`, `deliver()`, and `notify()` (extensible upcalls are handled using the generic handler). A node calls `forward()`

```
typedef int (*macedon_forward_handler)
   (char *msg, int size, int type,
    int nextHop, macedon_key nextHopKey);
typedef void (*macedon_deliver_handler)
   (char *msg, int size, int type);
typedef void (*macedon_notify_handler)
   (int type, int size, int *neighbors);
typedef int (*macedon_upcall_handler)
   (int operation, void *arg);
macedon_init(macedon_key bootstrap, int prot);
void macedon_register_handlers(
  macedon_forward_handler,macedon_deliver_handler,
  macedon_notify_handler,macedon_upcall_handler);
int macedon_create_group(macedon_key groupID);
void macedon_join(macedon_key groupID);
void macedon_leave(macedon_key groupID);
int macedon_route(macedon_key dest, char *msg,
     int size, int priority);
int macedon_multicast(macedon_key groupID,
     char *msg, int size, int priority);
int macedon_anycast(macedon_key groupID,
     char *msg, int size, int priority);
int macedon_routeIP(int dest, char *msg,
     int size, int priority);
```

Figure 3: Simplified MACEDON API.

once it makes a message routing decision. Intermediate nodes can change the message or its destination or quash the message altogether. The `notify()` upcall allows lower-layer protocols to inform higher layers of changes in neighbor lists (a higher layer may require this direct knowledge). An application optionally registers its upcall handlers with the `macedon_register_handlers()` function. At least one handler is necessary if the application is to receive any data through the overlay (having null handlers would be used when evaluating just the construction process of different overlays).

Figure 3 also shows `macedon_init()` that initializes an overlay identified by the application-specified well-known protocol value (akin to protocol values in IP). Once an application initializes and registers its handlers, it can send and receive data. For unicast data, the overlay must implement *routing* functionality that determines which neighbor receives data packets next. The `macedon_route()` function accepts a message and destination in the form of a macedon_key, meaning it is not necessarily an IP address (it could be a hash of an IP address or name). A similar primitive is `macedon_routeIP()` that enables native IP-based communication with an IP host.

Multicast primitives include `macedon_create_group()` to create sessions. Its sole input is the value, or handle, associated with the session (group). Receivers join and leave a session with `macedon_join()` and

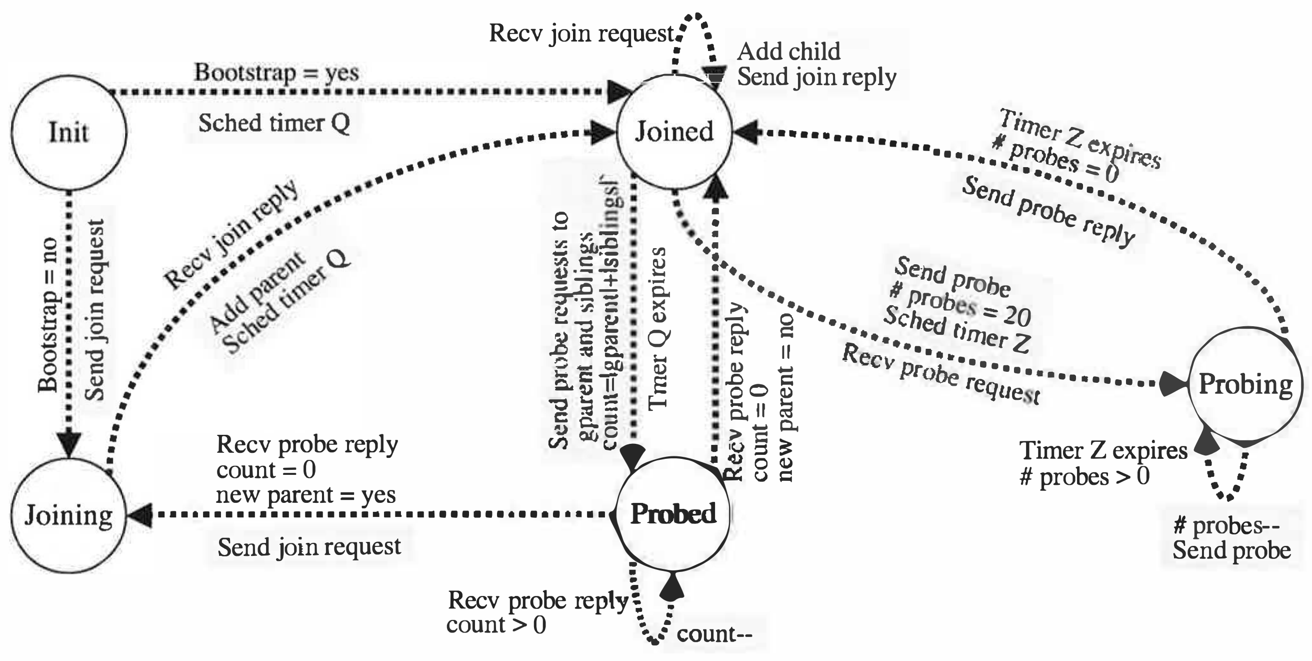


Figure 1: Portion of the Overcast algorithm representation. Circles represent FSM states. Directed edges identify transitions, with events as unshaded text and actions in shaded text boxes.

an overlay, it may transition from the joining phase and transmit a confirmation. Actions include scheduling a timer, transmitting a message, and changing node state. Overlays also employ periodic timers to execute overlay maintenance, check invariants, or some other periodic action. For example, Chord periodically checks and repairs its routing (finger) table entries.

Messages provide the fundamental mechanism for coordinating distributed actions and transmitting data. In Chord, a node transmits route repair requests to its neighbors. Chord nodes also transmit data messages through the overlay enabling application-to-application communication. Finally, an overlay protocol specifies which state variables change upon an event. In Chord, a node gathers route replies to determine if its route entries are stale, and if so, updates them accordingly.

2.1.4 An example: Overcast

An overlay's FSM behavior is specified in a *mac file*. To aid specification, it is helpful to first describe the high-level behavior of the algorithm graphically, as illustrated in Figure 1. The figure shows Overcast's [13] five system states and associated transitions. The protocol begins in the "init" state from which it transitions to the "joined" state if this node is the bootstrap node (i.e. the designated root of the overlay). Otherwise, it transmits a join request to the bootstrap. A joined node (including the bootstrap) receiving a join request will add the incoming node to its child neighbor list and transmit a join reply to confirm the process. Upon receiving the join reply, the joining node stores its new parent in its parent neighbor list and transitions to the "joined" state.

The Q timer allows joined nodes to periodically evaluate their position. When the timer expires, a node initiates probes from its grandparent and siblings (we omit the details of how a node acquires this information) and enters the "probed" state. It uses a state field to count the number of nodes probing it. Upon receiving a probe request, nodes send equally-spaced probes at some defined rate using the Z timer. Once all probes are transmitted, the probing node transmits the probe reply and returns to the "joined" state. After the probed node gathers the necessary replies from all nodes (count=0), it decides whether it should move to a new parent. If it moves, it again enters the "joining" state and sends a join request to the new parent. Otherwise, it simply returns to the "joined" state. Section 3 describes how this high-level representation is captured in the Overcast mac file.

2.2 The MACEDON API

Overlay algorithms typically target specific types of applications. An important characteristic of their implementation is the API they export. For example, a multicast overlay must export a send function to disseminate

2.1 FSM representation

Overlay nodes maintain local state regarding their current activities and communicate with neighbors through control messages. They use periodic timers to schedule future processing and may receive application commands instructing them to perform an operation. The fundamental premise of our approach is that these characteristics can be succinctly described by event-driven finite state machines (FSMs). In this model, *events* such as message reception, scheduled completion of timers, and application commands, trigger the overlay protocol to perform protocol *actions*. Actions include setting local node state, transmitting new messages, scheduling timers, and delivering application data, though this is not an exhaustive list. Events may occur nearly simultaneously, perhaps requiring the serialization of local state. In addition, events may cause the protocol to move from one *system state*, or phase of execution, to another. Behavior toward an event while in a certain system state may be different when in another state. In summary, we believe that we can sufficiently capture an overlay's intricate behavior by describing its system states, local node state, events, and the response to these events in this FSM framework. The following subsections describe the components of the MACEDON FSM abstraction.

2.1.1 Node state

Each overlay node maintains local state describing its current position and activities. Local state determines each node's relationship to current neighbors. For example, a tree-based overlay (e.g. Overcast) will have parent and children neighbors. The behavior of the node toward a peer may be different depending on its peer type. It may also maintain a list of *potential* peers to establish future peer relationships. This functionality is not required in certain overlays, when nodes either establish a peer relationship with or delete any knowledge of a potential peer. In other overlays, such as RON [2], such state may include a list of *all* nodes present in the overlay. Node state may also include specialized information that identifies characteristics about this node's position in the overlay. Examples include bandwidth estimates to neighbor nodes as in Overcast [13] and routing tables in DHTs. We term this type of node state *state variables*.

In addition, algorithms have system states that represent high-level phases of processing. For example, upon initialization, a node in the overlay may enter a"joining" phase where a join request message is transmitted to a node already in the overlay. A "probing" state could be where nodes probe a certain population of overlay participants, for instance, to reduce latency in the overlay.

2.1.2 Events

In our target systems, asynchronous events move a node from one system state to another, performing subsequent actions such as sending a message. Events include timer expirations, message reception, and API function calls. In message reception, a node processes the message, performing appropriate actions in response. For instance, a node receiving a join request may attempt to add the joining node as its neighbor.

When a scheduled timer expires, the node performs functions appropriate to this timer. For example, a NICE [4] node schedules timers to check protocol invariants. If a node cluster is unsuitably large or small, the node initiates a cluster split or merge. The node would change its system state and perform a number of coordinating actions. In another event, an application issues commands to the overlay. Upon receiving the call, the node may change its state and perform appropriate actions. Application commands fall into two categories, control commands for administrative operations and data commands for transmitting data through the overlay.

The distinction between control and data operations is central to MACEDON's handling of asynchronous events. Control operations modify node state and are exclusively serialized within a protocol instance. Data operations simply read node state, enabling shared protocol access. In MACEDON, events may occur simultaneously. For example, an application may spawn multiple threads, each of which can make an API call (control or data) into the overlay instance, thereby leading to potential race conditions. Likewise, multiple timer and transport threads may execute simultaneously. By allowing multiple data operations to proceed simultaneously, MACEDON exploits the advantages of multithreaded programming to achieve superior performance in delivering data through the overlay.

Overlay developers classify transitions as *control* (requiring write access to node state) or *data* (only read access). Using this classification, we determine the proper level of protocol instance locking on a per-transition basis. Each instance is secured with a read/write lock. Control operations secure the lock exclusively for writing, while data operations use read locking to allow multiple threads to execute in parallel, increasing performance when working threads block or a multi-processor is available.

2.1.3 Actions

A *transition*, representing a series of related actions, is uniquely identified by (event, FSM state). That is, the current system state determines an algorithm's response to specific events. For example, once a node has joined and queuing. While ns might address this shortcoming, it faces scalability limitations beyond a few hundred nodes, making overlay evaluation problematic. Live deployment certainly provides an existence proof, but does not enable evaluation under scale or highly dynamic conditions. The final phase, evaluation, involves processing the information generated through experimentation using hand-crafted tools. Researchers subsequently modify their implementations in light of code bugs or suboptimal performance. They employ disparate implementation techniques, causing the evaluation of competing overlays to reflect differences in implementation methodologies rather than in algorithmic principles and design.

To address these limitations, we present MACEDON, an infrastructure to simplify the design, development, evaluation, and comparison of large-scale overlays. In MACEDON, researchers specify algorithm behavior in terms of event-driven finite state machines (FSMs) consisting of system *states*, *events* (e.g. message reception, remote node failure, etc.), and *transitions* indicating the actions to take in response to events. From this high-level specification, MACEDON generates code for a variety of experimentation infrastructures leveraging shared (but extensible) libraries. The libraries implement much of the base overlay maintenance functionality, such as thread and timer management, network communication, debugging, and state serialization. As such, improvements in system support can be equally applied to all protocols. Ultimately, these system mechanisms enable fair comparisons of the merits of individual *algorithms* rather than artifacts of particular *implementations*.

MACEDON currently generates native C++ that runs unmodified in live Internet settings, including PlanetLab, and the ModelNet large-scale network emulator [27]. ModelNet enables us to subject overlays of thousands of nodes to the characteristics of large network topologies, capturing both scale and realism. MACEDON eliminates the need to maintain multiple versions of the same algorithm for different evaluation infrastructures. We provide built-in support for tracking popular overlay evaluation metrics, such as average delay penalty, communication overhead, and communication stretch. Our evaluation tools enable researchers to gain deeper understanding into the complex behavior of their algorithms, thus closing the streamlined development cycle.

To validate the utility of our approach, we have implemented of a number of overlays in the MACEDON framework. We have leveraged MACEDON to guide our design of AMMO [21] and Bullet [16]. Our MACEDON implementations also include Chord [25], NICE [4], Overcast [13], Pastry [22], Scribe [24] and SplitStream [6]. Here, we compare our generated code with published results and publicly available implementations. Our comparison indicates that MACEDON is able to reproduce or exceed performance of these systems, with concise system descriptions consisting of a few hundred lines. Using a standard API, applications and protocols coded to services of one overlay may easily switch to another providing similar functionality. For instance, the Scribe application-layer multicast protocol can be switched from using Pastry to Chord by changing a single line in its MACEDON specification. In isolation, our protocol implementations constitute an important contribution: the validation of results published separately by other authors. Taken together, they demonstrate the generality and utility of the MACEDON framework for developing and comparing overlays. Over time, we hope that code for a wide variety of overlays will become publicly available, further lowering the barrier to experiment with new ideas in this space.

This paper is organized as follows. We define overlays in terms of an abstraction in Section 2. Section 3 gives details of overlay implementation using the MACEDON language. We present validation of our methodology in Section 4. Section 5 compares MACEDON with other implementation infrastructures and describes other related work. We conclude with future work in Section 6.

2 Overlay Abstraction

We seek to provide a representation of distributed algorithms that is expressive enough to characterize the intricacies of different protocols, yet simple enough to facilitate implementation. To this end, we identify common characteristics of overlays and describe our FSM-based approach of describing them. We show how we use this unifying abstraction to enable concise descriptions of a wide array of network protocols. While it is not feasible to prove that all overlays share these characteristics, we have yet to encounter one that does not.

At a high level, an overlay network is a distributed algorithm where nodes establish logical *peer* or *neighbor* relationships with some subset of global participants forming a logical network *overlayed* atop the IP substrate. Examples include advanced communication semantics provided by multicast overlays and network maintenance as performed by BGP routers. A subset of these overlays export APIs that allow applications to transmit data through them. Our initial MACEDON implementation focuses on these algorithms, though we note that MACEDON is a generic framework for developing a wide variety of distributed systems. In particular, our initial work targets distributed hash tables (DHTs) and application-level multicast, described further in Section 5.

MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks*

Adolfo Rodriguez, Charles Killian, Sooraj Bhat [†], Dejan Kostić
Duke University
{razor,ckillian,dkostic}@cs.duke.edu

Amin Vahdat [‡]
UC San Diego
vahdat@cs.ucsd.edu

Abstract

Currently, researchers designing and implementing large-scale overlay services employ disparate techniques at each stage in the production cycle: design, implementation, experimentation, and evaluation. As a result, complex and tedious tasks are often duplicated leading to ineffective resource use and difficulty in fairly comparing competing algorithms. In this paper, we present MACEDON, an infrastructure that provides facilities to: i) specify distributed algorithms in a concise domain-specific language; ii) generate code that executes in popular evaluation infrastructures and in live networks; iii) leverage an overlay-generic API to simplify the interoperability of algorithm implementations and applications; and iv) enable consistent experimental evaluation. We have used MACEDON to implement and evaluate a number of algorithms, including AMMO, Bullet, Chord, NICE, Overcast, Pastry, Scribe, and SplitStream, typically with only a few hundred lines of MACEDON code. Using our infrastructure, we are able to accurately reproduce or exceed published results and behavior demonstrated by current publicly available implementations.

1 Introduction

Designing and implementing robust, high-performance networked systems is difficult. Overcoming this difficulty is increasingly important as an ever larger fraction of the world's infrastructure comes to rely upon networked systems. Challenges include network and host failures, highly variable communication patterns, race conditions, reproducing bugs, and security. While the advent of higher-level programming languages such as Java has raised the level of abstraction and somewhat eased this burden, most programmers are still faced with the daunting task of re-inventing appropriate techniques for dealing with asynchronous, failure-prone network environments known by a handful of elite programmers.

We seek to explore the appropriate programming models and development environments with the twin goals of: i) making it easier to advance the state of the art in building robust networked systems, and ii) bringing this state of the art to programmers at large. While this is an ambitious effort, this paper attempts to uncover some of the relevant issues by focusing on programming language and runtime support for designing, implementing, and evaluating an emerging class of distributed services, overlay networks. We initially focus on a few types of overlays (distributed hash tables, DHTs, [22, 25, 30] and application-layer multicast [6, 9, 12, 16, 23, 24, 31]), though we believe our framework is applicable to other classes of overlays such as indirect routing (e.g. RON [2]), 6Bone, and BGP.

We view current overlay research as following a cycle consisting of four phases, each of which suffers from a number of challenges. First, an overlay researcher designs an algorithm that optimizes for network metrics such as latency and provides for application behavior such as $O(\lg n)$ routing hops in DHTs. In the second phase, one or more implementations are created to evaluate algorithm performance. For example, many researchers create hand-crafted simulators for evaluating performance under scale and live implementations for evaluation in real settings. Creating such implementations is often tedious and difficult, both due to the size of software components needed and the complexity of such functionality.

Using an algorithm's implementations, researchers use experimentation to gather run-time performance data in the third phase. Usually, this includes both simulation (such as with the network simulator, ns [29]) and small-scale live Internet runs (e.g. PlanetLab [19]). Unfortunately, custom simulation does not capture the full intricacies of network behavior such as congestion

*This research is supported in part by the National Science Foundation (EIA-9972879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Additional information is available on the MACEDON website: http://www.cs.duke.edu/~razor/MACEDON.

†Now a student at Georgia Institute of Technology, sooraj@cc.gatech.edu

‡Supported by NSF CAREER award (CCR-9984328).

[12] J. Dike. User-mode Linux. In *Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, CA, Nov 2001.

[13] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proc. SIGCOMM '96*, pages 53–59, Stanford, CA, Aug 1996.

[14] Ensim Corp. Ensim Virtual Private Server. `http://www.ensim.com/products/materials/datasheet_vps_051003.pdf%`, 2000.

[15] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1; RFC 2616. *Internet Req. for Cmts.*, Jun 1999.

[16] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf`, Jun 2002.

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proc. 19th SOSP*, Lake George, NY, Oct 2003.

[18] N. Hardy. The KeyKOS Architecture. *Operating Systems Review*, 19(4):8–25, Oct 1985.

[19] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, Mar 2002.

[20] C. Jin, D. Wei, S. H. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cottrell, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, and S. Singh. FAST TCP: From Theory to Experiments, Apr 2003. Submitted for publication.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.*, Maastricht, The Netherlands, May 2000.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. 9th ASPLOS*, pages 190–201, Cambridge, MA, Nov 2000.

[23] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J. Sel. Areas Comm.*, 14(7):1280–1297, 1996.

[24] Linux Advanced Routing and Traffic Control. `http://lartc.org/`.

[25] Linux VServers Project. `http://linux-vserver.org/`.

[26] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th ICDCS*, pages 104–111, San Jose, CA, Jun 1988.

[27] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Int. Conf. Multimedia Computing & Systems*, pages 90–99, Boston, MA, May 1994.

[28] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. 2nd OSDI*, pages 153–167, Seattle, WA, Oct 1996.

[29] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An Architecture for Enabling Sensor-Enriched Internet Service. Technical Report IRP–TR–03–04, Intel Research Pittsburgh, Jun 2003.

[30] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I*, Princeton, NJ, Oct 2002.

[31] L. Peterson, J. Hartman, S. Muir, T. Roscoe, and M. Bowman. Evolving the Slice Abstraction, Jan 2004. `http://www.planet-lab.org/`.

[32] Plkmod: SILK in PlanetLab. `http://www.cs.princeton.edu/~acb/plkmod`.

[33] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. INFOCOM 2002*, pages 1190–1199, New York City, Jun 2002.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. 18th Middleware*, pages 329–350, Heidelberg, Germany, Nov 2001.

[35] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proc. 18th SOSP*, pages 188–201, Banff, Alberta, Canada, Oct 2001.

[36] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed internet measurement. In *Proc. 4th USITS*, Seattle, WA, Mar 2003.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *Proc. SIGCOMM 2001*, pages 149–160, San Diego, CA, Aug 2001.

[38] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: Offering Internet QoS Using Overlays. In *Proc. HotNets–I*, Princeton, NJ, Oct 2002.

[39] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proc. 5th OSDI*, pages 345–360, Boston, MA, Dec 2002.

[40] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proc. HotNets–II*, Cambridge, MA, Nov 2003.

[41] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI*, pages 195–209, Boston, MA, December 2002.

[42] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th OSDI*, pages 255–270, Boston, MA, Dec 2002.

[43] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proc. 11th ICNP*, Atlanta, GA, Nov 2003.

3. the slice creation service performs a node manager bind RPC to bind the ticket to a new slice name;

4. the node manager, after completing the RPCs, creates the new vserver and notifies the necessary resource schedulers to effect the newly added resource bindings for the new slice; and

5. the node manager calls vadduser to instantiate the vserver and then calls vserver-init to start execution of software within the new vserver.

Running on a 1.2GHz Pentium, the first three steps complete in 0.15 seconds, on average. How long the fourth and fifth steps takes depends on how the user wants to initialize the slice. At a minimum, the vserver creation and initialization takes an additional 9.66 seconds on average. However, this does not include the time to load and initialize any service software such as `sshd` or other packages. It also assumes a hit in a warm cache of vservers. Creating a new vserver from scratch takes over a minute.

5.3 Service Initialization

This section uses an example service, Sophia [40], to demonstrate how long it takes to initialize a service once a slice exists. Sophia's slice is managed by a combination of RPM, `apt-get`, and custom slice tools. When a Sophia slice is created, it must be loaded with the appropriate environment. This is accomplished by executing a boot script inside each vserver. This script downloads and installs the `apt-get` tools and a root Sophia slice RPM, and starts an update process. Using `apt-get`, the update process periodically downloads the tree of current packages specific for the Sophia slice. If a newer package based on the RPM hierarchy is found, it and its dependencies are download and installed. With this mechanism, the new versions of packages are not directly pushed to all the nodes, but are published in the Sophia packages tree. The slice's update mechanism then polls (potentially followed with an action request push) the package tree and performs the upgrade actions.

In the current setting, it takes on average 11.2 seconds to perform an empty update on a node; i.e., to download the package tree, but not find anything new to upgrade. When a new Sophia "core" package is found and needs to be upgraded, the time increases to 25.9 seconds per node. These operations occur in parallel, so the slice upgrade time is not bound by the sum of node update times. However, the slice is to be considered upgraded only when all of its active nodes are finished upgrading. When run on 180 nodes, the average update time (corresponding to the slowest node) is 228.0 seconds. The performance could be much improved, for example, by using a better distribution mechanism. Also, a faster alternative to the RPM package dependencies system could improve the locally performed dependency checks.

6 Conclusions

Based on experience providing the network research community with a platform for planetary-scale services, the PlanetLab OS has evolved a set of mechanisms to support distributed virtualization and unbundled management. The design allows network services to run in a slice of PlanetLab's global resources, with the PlanetLab OS providing only local (per-node) abstractions and as much global (network-wide) functionality as possible pushed onto infrastructure services running in their own slices. Only slice creation (coupled with resource allocation) and slice termination run as a global privileged service, but in the long-term, we expect a set of alternative infrastructure services to emerge and supplant these bootstrap services.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th SOSP*, pages 131–145, Banff, Alberta, Canada, Oct 2001.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proc. Pervasive 2002*, pages 195–210, Zurich, Switzerland, Aug 2002.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd OSDI*, pages 45–58, New Orleans, LA, Feb 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th SOSP*, Lake George, NY., Oct 2003.

[5] A. Bavier, T. Voigt, M. Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report 2002–009, Department of Information Technology, Uppsala University, Uppsala, Sweden, Feb 2002.

[6] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proc. IEEE Conf. Comp. Networks*, Nov 1997.

[7] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proc. 4th USENIX Windows Sys. Symp.*, pages 13–24, Seattle, WA, Aug 2000.

[8] Y. Chu, S. Rao, and H. Zhang. A Case For End System Multicast. In *Proc. SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, Jun 2000.

[9] B. Chun, J. Lee, and H. Weatherspoon. Netbait: a Distributed Worm Detection Service, 2002. `http://netbait.planet-lab.org/`.

[10] B. Chun and T. Spalink. Slice Creation and Management, Jun 2003. `http://www.planet-lab.org/`.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, pages 202–215, Banff, Alberta, Canada, Oct 2001.

node that summarize the traffic sent in the last hour by IP destination and slice name. The hope is that an administrator at a site that receives questionable packets from a PlanetLab machine will type the machine's name or IP address into his or her browser, find the audit-generated pages, and use them to contact the experimenters about the traffic. For example, an admin who clicks on an IP address in the destination summary page is shown all of the PlanetLab accounts that sent a packet to that address within the last hour, and provided with links to send email to the researchers associated with these accounts. Another link directs the admin to the network traffic database at `www.planet-lab.org` where back logs are archived, so that he or she can make queries about the origin of traffic sent earlier than one hour ago.

Our experience with the traffic auditing service has been mixed. On the one hand, the PlanetLab support team has found it very useful for responding to traffic queries: after receiving a complaint, they use the Web interface to identify the responsible party and forward the complaint on to him. As a result, there has been a reduction in overall incident response time and the time invested by support staff per incident. On the other hand, many external site administrators either do not find the web page or choose not to use it. For example, when receiving a strange packet from `planetlab-1.cs.princeton.edu`, most sites respond by sending email to `abuse@princeton.edu`; by the time the support team receives the complaint, it has bounced through several levels of university administration. We may be able to avoid this indirection by providing reverse DNS mappings for all nodes to *nodename*`.planet-lab.org`, but this requires effort from each site that sponsors PlanetLab nodes. Finding mechanisms that further decentralize the problem-response process is ongoing work.

Finally, although our experience to date has involved implementing and querying read-only sensors that can be safely accessed by untrusted monitoring services, one could easily imagine PlanetLab also supporting a set of *actuators* that only trusted management services could use to control PlanetLab. For example, there might be an actuator that terminates a slice, so that a Sophia expression can be written to kill a slice that has violated global bandwidth consumption limits. Today, slice termination is not exposed as an actuator, but is implemented in the node manager; it can be invoked only by the trusted PLC service, or an authenticated network operator that remotely logs into the node manager.

5 Evaluation

This section evaluates three aspects of slice creation and initialization.

5.1 Vserver Scalability

The scalability of vservers is primarily determined by disk space for vserver root filesystems and service-specific storage. On PlanetLab, each vserver is created with a root filesystem that points back to a trimmed-down reference root filesystem which comprises 1408 directories and 28003 files covering 508 MB of disk. Using vserver's primitive COW on all files, excluding those in `/etc` and `/var`, each vserver root filesystem mirrors the reference root filesystem while only requiring 29 MB of disk space, 5.7% of the original root filesystem size. This 29 MB consists of 17.5 MB for a copy of `/var`, 5.6 MB for a copy of `/etc`, and 5.9 MB to create 1408 directories (4 KB per directory). Given the reduction in vserver disk footprints afforded by COW, we have been able to create 1,000 vservers on a single PlanetLab node. In the future, we would like to push disk space sharing even further by using a true filesystem COW and applying techniques from systems such as the Windows Single Instance Store [7].

Kernel resource limits are a secondary factor in the scalability of vservers. While each vserver is provided with the illusion of its virtual execution environment, there still remains a single copy of the underlying operating system and associated kernel resources. Under heavy degrees of concurrent vserver activity, it is possible that limits on kernel resources may become exposed and consequently limit system scalability. (We have already observed this with file descriptors.) The nature of such limits, however, are no different from that of large degrees of concurrency or resource usage within a single vserver or even on an unmodified Linux kernel. In both cases, one solution is to simply extend kernel resource limits by recompiling the kernel. Of course, simple scaling up of kernel resources may be insufficient if inefficient algorithms are employed within the kernel (e.g., $O(n)$ searches on linked lists). Thus far, we have yet to run into these types of algorithmic bottlenecks.

5.2 Slice Creation

This section reports how long it takes the node manager to create a vserver on a single node. The current implementation of PLC has each node poll for slice creation instructions every 10 minutes, but this is an artifact of piggybacking the slice creation mechanism on existing software update machinery. It remains to be seen how rapidly a slice can be deployed on a large number of nodes.

To create a new slice on a specific node, a slice creation service must complete the following steps at that node:

1. the slice creation service contacts a port mapping service to find the port where the node manager's XML-RPC server is listening for requests;

2. the slice creation service performs a node manager acquire RPC to obtain an rcap for immediate rights to a vserver and best-effort resource usage;

vide a shared interface to alternative monitoring services.

4.4.1 Interface

A sensor provides pieces of information that are available (or can be derived) on a local node. A *sensor server* aggregates several sensors at a single access point, thereby providing controlled sharing of sensors among many clients (e.g., monitoring services). To obtain a sensor reading, a client makes a request to a sensor server. Each sensor outputs one or more tuples of untyped data values. Every tuple from a sensor conforms to the same schema. Thus, a sensor can be thought of as providing access to a (potentially infinite) database table.

Sensor semantics are divided into two types: *snapshot* and *streaming*. Snapshot sensors maintain a finite size table of tuples, and immediately return the table (or some subset of it) when queried. This can range from a single tuple which rarely varies (e.g. "number of processors on this machine") to a circular buffer that is constantly updated, of which a snapshot is available to clients (for instance, "the times of 100 most recent connect system calls, together with the associated slices"). Streaming sensors follow an event model, and deliver their data asynchronously, a tuple at a time, as it becomes available. A client connects to a streaming sensor and receives tuples until either it or the sensor server closes the connection.

More precisely, a sensor server is an HTTP [15] compliant server implementing a subset of the specification (GET and HEAD methods only) listening to requests from `localhost` on a particular port. Requests come in the form of uniform resource identifiers (URIs) in GET methods. For example, the URI:

```
http://localhost:33080/nodes/ip/name
```

is a request to the sensor named "`nodes`" at the sensor server listening on port 33080. The portion of the URI after the sensor name (i.e., `ip/name`) is interpreted by the sensor. In this case, the nodes sensor returns comma-separated lines containing the IP address and DNS name of each registered PlanetLab node. We selected HTTP as the sensor server protocol because it is a straightforward and well-supported protocol. The primary format for the data returned by the sensor is a text file containing easy-to-parse comma separated values.

4.4.2 Implementation

An assortment of sensor servers have been implemented to date, all of which consist of a stripped-down HTTP server encapsulating an existing source of information. For example, one sensor server reports various information about kernel activities. The sensors exported by this server are essentially wrappers around the `/proc` file system. Example sensors include `meminfo` (returns information about current memory usage), `load` (returns 1-minute load average), `load5` (returns 5-minute load average), `uptime` (returns uptime of the node in seconds), and `bandwidth(slice)` (returns the bandwidth consumed by a `slice`, given by a slice id).

These examples are simple in at least two respects. First, they require virtually no processing; they simply parse and filter values already available in `/proc`. Second, they neither stream information nor do they maintain any history. One could easily imagine a variant of `bandwidth`, for example, that both streams the bandwidth consumed by the slice over that last 5 minute period, updated once every five minutes, or returns a table of the last *n* readings it had made.

In contrast, a more complex sensor server will shortly become available that reports information about how the local host is connected to the Internet, including path information returned by `traceroute`, peering relationships determined by a local BGP feed, and latency information returned by `ping`. This sensor server illustrates that some sensors may be expensive to invoke, possibly sending and receiving messages over the Internet before they can respond, and as a result may cache the results of earlier invocations.

4.4.3 Discussion

Using the sensor abstraction, and an emerging collection of sensor implementations, an assortment of monitoring services are being deployed. Many of these services are modeled as distributed query processors, including PIER [19], Sophia [40], and IrisNet [29].

The long-term goal is for these monitoring services to detect, reason about, and react to anomalous behavior before it becomes disruptive. However, PlanetLab has an immediate need of responding to disruptions after the fact. Frequently within the past year, traffic generated by PlanetLab researchers has caught the attention of ISPs, academic institutions, Web sites, and sometimes even home users. In nearly all cases, the problems have stemmed from naïve service design and analysis, programmer errors, or hyper-sensitive intrusion detection systems. Examples include network mapping experiments that probe large numbers of random IP addresses (looks like a scanning worm), services aggressively `traceroute`'ing to certain target sites on different ports (looks like a portscan), services performing distributed measurement to a target site (looks like a DDoS attack), services sending large numbers of ICMP packets (not a bandwidth problem, but renders low-end routers unusable), and so on. Addressing such complaints requires an *auditing* tool that can map an incident onto a responsible party.

Specifically, a traffic auditing service runs on every PlanetLab node, snooping all outgoing traffic using the administrative raw sniffer socket provided by the SILK module that tags each packet with the ID of the sending vserver. From each packet, the auditing service logs the time it was sent, the IP source and destination, protocol, port numbers, and TCP flags if applicable. It then generates Web pages on each

16-bit field present in the ICMP header). Only messages containing the bound identifier can be received and sent through a safe raw ICMP socket.

PlanetLab users can debug protocol implementations or applications using "sniffer" raw sockets. Most slices lack the necessary capability to put the network card into promiscuous mode and so cannot run `tcpdump` in the standard way. A sniffer raw socket can be bound to a TCP or UDP port that was previously opened in the same vserver; the socket receives copies of all packets sent or received on the port but cannot send packets. A utility called `plabdump` opens a sniffer socket and pipes the packets to `tcpdump` for parsing, so that a user can get full tcpdump-style output for any of his or her connections.

4.3.2 Implementation

Safe raw sockets are implemented by the SILK kernel module, which intercepts all incoming IP packets using Linux's `netfilter` interface and demultiplexes each to a Linux socket or to a safe raw socket. Those packets that demultiplex to a Linux socket are returned to Linux's protocol stack for further processing; those that demultiplex to a safe raw socket are placed directly in the per-socket queue maintained by SILK. When a packet is sent on a safe raw socket, SILK intercepts it by wrapping the socket's `sendmsg` function in the kernel and verifies that the addresses, protocol, and port numbers in the packet headers are correct. If the packet passes these checks, it is handed off to the Linux protocol stack via the standard raw socket `sendmsg` routine.

SILK's port manager maintains a mapping of port assignments to vservers that serves three purposes. First, it ensures that the same port is not opened simultaneously by a TCP/UDP socket and a safe raw socket (sniffer sockets excluded). To implement this, SILK must wrap the `bind`, `connect`, and `sendmsg` functions of standard TCP/UDP sockets in the kernel, so that an error can be returned if an attempt is made to bind to a local TCP or UDP port already in use by a safe raw socket. In other words, SILK's port manager must approve or deny *all* requests to bind to a port, not just those of safe raw sockets. Second, when `bind` is called on a sniffer socket, the port manager can verify that the port is either free or already opened by the vserver attempting the bind. If the port was free, then after the sniffer socket is bound to it the port is owned by that vserver and only that vserver can open a socket on that port. Third, SILK allows the node manager described in Section 4.2.1 to reserve specific ports for the use of a particular vserver. The port manager stores a mapping for the reserved port so that it is considered owned by that vserver, and all attempts by other vservers to bind to that port will fail.

4.3.3 Discussion

The driving application for safe raw sockets has been the Scriptroute [36] network measurement service. Scriptroute provides users with the ability to execute measurement scripts that send arbitrary IP packets, and was originally written to use privileged raw sockets. For example, Scriptroute implements its own versions of `ping` and `traceroute`, and so needs to send ICMP packets and UDP packets with the IP TTL field set. Scriptroute also requires the ability to generate TCP SYN packets containing data to perform `sprobe`-style bottleneck bandwidth estimation. Safe raw sockets allowed Scriptroute to be quickly ported to PlanetLab by simply adding a few calls to `bind`. Other users of safe raw sockets are the modified versions of `traceroute` and `ping` that run in a vserver (on Linux, these utilities typically run with root privileges in order to open a raw socket). Safe raw sockets have also been used to implement user-level protocol stacks, such as variants of TCP tuned for high-bandwidth pipes [20], or packet re-ordering when striping across multiple overlay paths [43]. A BSD-based TCP library currently runs on PlanetLab.

Safe raw sockets are just one example of how PlanetLab services need to be able to share certain address spaces. Another emerging example is that some slices want to customize the routing table so as to control IP tunneling for their own packets. Yet another example is the need to share access to well-known ports; e.g., multiple services want to run DNS servers. In the first case, we are adopting an approach similar to that used for raw sockets: partition the address space by doing early demultiplexing at a low level in the kernel. In the second case, we plan to implement a user-level demultiplexor. In neither case is a single service granted privileged and exclusive access to the address space.

4.4 Monitoring

Good monitoring tools are clearly required to support a distributed infrastructure such as PlanetLab, which runs on hundreds of machines worldwide and hosts dozens of network services that use and interact with each other and the Internet in complex and unpredictable ways. Managing this infrastructure—collecting, storing, propagating, aggregating, discovering, and reacting to observations about the system's current conditions—is one of the most difficult challenges facing PlanetLab.

Consistent with the principle of unbundled management, we have defined a low-level *sensor interface* for uniformly exporting data from the underlying OS and network, as well as from individual services. Data exported from a sensor can be as simple as the process load average on a node or as complex as a peering map of autonomous systems obtained from the local BGP tables. That is, sensors encapsulate raw observations that already exist in many different forms, and pro-manager on each node periodically communicates with the central PLC server to obtain policy about what slices can be created and how many resources are to be bound to each.

PLC allocates resources to participating institutions and their projects, but it also allocates resources to other brokerage services for redistribution. In this way, PLC serves as a bootstrap brokerage service, where we expect more sophisticated (and more decentralized) services to evolve over time. We envision this evolution proceeding along two dimensions.

First, PLC currently allows the resources bound to a slice to be specified by a simple (share, duration) pair, rather than exposing the more detailed rspec accepted by the node manager. The share specifies a relative share of each node's CPU and link capacity that the slice may consume, while duration indicates the period of time for which this allocation is valid. PLC policy specifies how to translate the share value given by a user into a valid rspec presented to the node manager. Over time we expect PLC to expose more of the rspec structure directly to users, imposing less policy on resource allocation decisions. We decided to initially hide the node manager interface because its semantics are not well-defined at this point: how to divide resources into allocatable units is an open problem, and to compensate for this difficulty, the fields of the rspec are meaningful only to the individual schedulers on each node.

Second, there is significant interest in developing market-based brokerage services that establish resource value based on demand, and a site's purchasing capacity based on the resources it contributes. PLC currently has a simple model in which each site receives an equal number of shares that it redistributes to projects at that site. PLC also allocates shares directly to certain infrastructure services, including experimental brokerage services that are allowed to redistribute them to other services. In the long term, we envision each site administrator deciding to employ different, or possibly even multiple, brokerage services, giving each some fraction of its total capacity. (In effect, site administrators currently allocate 100% of their resources to PLC by default.)

To date, PLC supports two additional brokerage services: Emulab and SHARP. Emulab supports short-lasting network experiments by pooling a set of PlanetLab resources and establishing a batch queue that slices are able to use to serialize their access. This is useful during times of heavy demand, such as before conference deadlines. SHARP [17] is a secure distributed resource management framework that allows agents, acting on behalf of sites, to exchange computational resources in a secure, fully decentralized fashion. In SHARP, agents peer to trade resources with peering partners using cryptographically signed statements about resources.

4.3 Network Virtualization

The PlanetLab OS supports network virtualization by providing a "safe" version of Linux raw sockets that services can use to send and receive IP packets without root privileges. These sockets are safe in two respects. First, each raw socket is bound to a particular TCP or UDP port and receives traffic only on that port; conflicts are avoided by ensuring that only one socket of any type (i.e., standard TCP/UDP or raw) is sending on a particular port. Second, outgoing packets are filtered to make sure that the local addresses in the headers match the binding of the socket. Safe raw sockets support network measurement experiments and protocol development on PlanetLab.

Safe raw sockets can also be used to monitor traffic within a slice. A "sniffer" socket can be bound to any port that is already opened by the same VM, and this socket receives copies of all packet headers sent and received on that port. Additionally, sufficiently privileged slices can open a special administrative sniffer socket that receives copies of all outgoing packets on the machine tagged with the context ID of the sending vserver; this administrative socket is used to implement the traffic monitoring facility described in Section 4.4.

4.3.1 Interface

A standard Linux raw socket captures all incoming IP packets and allows writing of arbitrary packets to the network. In contrast, a safe raw socket is bound to a specific UDP or TCP port and receives only packets matching the protocol and port to which it is bound. Outgoing packets are filtered to ensure that they are well-formed and that source IP and UDP/TCP port numbers are not spoofed.

Safe raw sockets use the standard Linux socket API with minor semantic differences. Just as in standard Linux, first a raw socket must be created with the `socket` system call, with the difference that it is necessary to specify `IPPROTO_TCP` or `IPPROTO_UDP` in the protocol field. It must then be bound to a particular local port of the specified protocol using the Linux `bind` system call. At this point the socket can send and receive data using the usual `sendto`, `sendmsg`, `recvfrom`, `recvmsg`, and `select` calls. The data received includes the IP and TCP/UDP headers, but not the link layer header. The data sent, by default, does not need to include the IP header; a slice that wants to include the IP header sets the `IP_HDRINCL` socket option on the socket.

ICMP packets can also be sent and received through safe raw sockets. Each safe raw ICMP socket is bound to either a local TCP/UDP port or an ICMP identifier, depending on the type of ICMP messages the socket will receive and send. To get ICMP packets associated with a specific local TCP/UDP port (e.g., Destination Unreachable, Source Quench, Redirect, Time Exceeded, Parameter Problem), the ICMP socket needs to be bound to the specific port. To exchange ICMP messages that are not associated with a specific TCP/UDP port—e.g., Echo, Echo Reply, Timestamp, Timestamp Reply, Information Request, and Information Reply—the socket has to be bound to a specific ICMP identifier (a to a virtual machine (vserver) at slice creation time using the following operation:

```
bind(slice_name, rcap)
```

This operation takes a slice identifier and an rcap as arguments, and assuming the rspec associated with the rcap includes the right to create a virtual machine, creates the vserver and binds the resources described by the rcap to it. Note that the node manager supports other operations to manipulate rcaps, as described more fully elsewhere [10].

4.2.2 Implementation

Non-renewable resources, such as memory pages, disk space, and file descriptors, are isolated using per-slice reservations and limits. These are implemented by wrapping the appropriate system calls or kernel functions to intercept allocation requests. Each request is either accepted or denied based on the slice's current overall usage, and if it is accepted, the slice's counter is incremented by the appropriate amount.

For renewable resources such as CPU cycles and link bandwidth, the OS supports two approaches to providing isolation: *fairness* and *guarantees*. Fairness ensures that each of the N slices running on a node receives no less than $1/N$ of the available resources during periods of contention, while guarantees provide a slice with a reserved amount of the resource (e.g., 1Mbps of link bandwidth). PlanetLab provides CPU and bandwidth guarantees for slices that request them, and "fair best effort" service for the rest. In addition to isolating slices from each other, resource limits on outgoing traffic and CPU usage can protect the rest of the world from PlanetLab.

The Hierarchical Token Bucket (htb) queuing discipline of the Linux Traffic Control facility (tc) [24] is used to cap the total outgoing bandwidth of a node, cap per-vserver output, and to provide bandwidth guarantees and fair service among vservers. The node administrator configures the root token bucket with the maximum rate at which the site is willing to allow traffic to leave the node. At vserver startup, a token bucket is created that is a child of the root token bucket; if the service requests a guaranteed bandwidth rate, the token bucket is configured with this rate, otherwise it is given a minimal rate (5Kbps) for "fair best effort" service. Packets sent by a vserver are tagged in the kernel and subsequently classified to the vserver's token bucket. The htb queuing discipline then provides each child token bucket with its configured rate, and fairly distributes the excess capacity from the root to the children that can use it in proportion to their rates. A bandwidth cap can be placed on each vserver limiting the amount of excess capacity that it is able to use. By default, the rate of the root token bucket is set at 100Mbps; each vserver is capped at 10Mbps and given a rate of 5Kbps for "fair best effort" service.

In addition to this general rate-limiting facility, htb can also be used to limit the outgoing rate for certain classes of packets that may raise alarms within the network. For instance, we are able to limit the rate of outgoing pings (as well as packets containing IP options) to a small number per second; this simply involves creating additional child token buckets and classifying outgoing packets so that they end up in the correct bucket. Identifying potentially troublesome packets and determining reasonable output rates for them is a subject of ongoing work.

CPU scheduling is implemented by the SILK kernel module, which leverages Scout [28] to provide vservers with CPU guarantees and fairness. Replacing Linux's CPU scheduler was necessary because, while Linux provides approximate fairness between individual processes, it cannot enforce fairness between vservers; nor can it provide guarantees. PlanetLab's CPU scheduler uses a proportional sharing (PS) scheduling policy to fairly share the CPU. It incorporates the resource container [3] abstraction and maps each vserver onto a resource container that possesses some number of shares. Individual processes spawned by the vserver are all placed within the vserver's resource container. The result is that the vserver's set of processes receives a CPU rate proportional to the vserver's shares divided by the sum of shares of all active vservers. For example, if a vserver is assigned 10 shares and the sum of shares of all active vservers (i.e., vservers that contain a runnable process) is 50, then the vserver with 10 shares gets $10/50 = 20\%$ of the CPU.

The PS scheduling policy is also used to provide minimum cycle guarantees by capping the number of shares and using an admission controller to ensure that the cap is not exceeded. The current policy limits the number of outstanding CPU shares to 1000, meaning that each share is a guarantee for at least 0.1% of the CPU. Additionally, the PlanetLab CPU scheduler provides a switch to allow a vserver to proportionally share the excess capacity, or to limit it to its guaranteed rate (similar to the Nemesis scheduler [23]). In the previous example, the vserver with 10 shares received 20% of the CPU because it was allowed to proportionally share the excess; with this bit turned off, it would be rate-capped at $10/1000 = 1\%$ of the CPU.

4.2.3 Discussion

We have implemented a centrally-controlled brokerage service, called PlanetLab Central (PLC), that is responsible for globally creating slices [31]. PLC maintains a database of principals, slices, resource allocations, and policies on a central server. It exports an interface that includes operations to create and delete slices; specify a boot script, set of user keys, and resources to be associated with the slice; and instantiate a slice on a set of nodes. A per-node component of PLC, called a *resource manager*, runs in a privileged slice; it is allowed to call the acquire operation on each node. The resource on-write memory segments across unrelated vservers. Disk space sharing is achieved using the filesystem immutable invert bit which allows for a primitive form of copy-on-write (COW). Using COW on chrooted vserver root filesystems, vserver disk footprints are just 5.7% of that required with full copies (Section 5.1). Comparable sharing in a virtual machine monitor or isolation kernel is strictly harder, although with different isolation guarantees.

PlanetLab's application of vservers makes extensive use of the Linux capability mechanism. Capabilities determine whether privileged operations such as pinning physical memory or rebooting are allowed. In general, the vserver `root` account is denied all capabilities that could undermine the security of the machine (e.g., accessing raw devices) and granted all other capabilities. However, as discussed in Section 2.3, each PlanetLab node supports two special contexts with additional capabilities: the *node manager* and the *admin slice*.

The node manager context runs with standard `root` capabilities and includes the machinery to create a slice, initialize its state, and assign resources to it; sensors that export information about the node; and a traffic auditing service. The admin slice provides weaker privileges to site administrators, giving them a set of tools to manage the nodes without providing full `root` access. The admin context has a complete view of the machine and can, for example, cap the node's total outgoing bandwidth rate, kill arbitrary processes, and run `tcpdump` to monitor traffic on the local network.

4.1.3 Discussion

Virtualizing above the kernel has a cost: weaker guarantees on isolation and challenges for eliminating QoS crosstalk. Unlike virtual machine monitors and isolation kernels that provide isolation at a low level, vservers implement isolation at the system call interface. Hence, a malicious vserver that exploits a Linux kernel vulnerability might gain control of the operating system, and hence compromise security of the machine. We have observed such an incident, in which a subset of PlanetLab nodes were compromised in this way. This would have been less likely using a lower-level VM monitor. Another potential cost incurred by virtualizing above the kernel is QoS crosstalk. Eliminating all QoS crosstalk (e.g., interactions through the buffer cache) is strictly harder with vservers. As described in the next section, however, fairly deep isolation can be achieved.

The combination of vservers and capabilities provides more flexibility in access control than we currently use in PlanetLab. For example, sensor slices (see section 4.4) could be granted access to information sources that cannot otherwise easily be shared among clients. As we gain experience on the privileges services actually require, extending the set of Linux capabilities is a natural path toward exposing privileged operations in a controlled way.

4.2 Isolation and Resource Allocation

A key feature of slices is the isolation they provide between services. Early experience with PlanetLab illustrates the need for slice isolation. For example, we have seen slices acquire all available file descriptors on several nodes, preventing other slices from using the disk or network; routinely fill all available disk capacity with unbounded event logging; and consume 100% of the CPU on 50 nodes by running an infinite loop. Isolating slices is necessary to make the platform useful.

The node manager provides a low-level interface for obtaining resources on a node and binding them to a local VM that belongs to some slice. The node manager does not make any policy decisions regarding how resources are allocated, nor is it remotely accessible. Instead, a bootstrap brokerage service running in a privileged slice implements the resource allocation policy. This policy includes how many resources to allocate to slices that run other brokerage services; such services are then free to redistribute those resources to still other slices. Note that resource allocation is largely controlled by a central policy in the current system, although we expect it to eventually be defined by the node owner's local administrator. Today, the only resource-related policy set by the local node administrator is an upper bound on the total outgoing bandwidth that may be consumed by the node.

4.2.1 Interface

The node manager denotes the right to use a set of node resources as a *resource capability* (rcap)—a 128-bit opaque value, the knowledge of which provides access to the associated resources. The node manager provides privileged slices with the following operation to create a resource capability:

$$\text{rcap} \leftarrow \text{acquire(rspec)}$$

This operation takes a *resource specification* (rspec) as an argument, and returns an rcap should there by sufficient resources on the node to satisfy the rspec. Each node manager tracks both the set of resources available on the node, and the mapping between committed resources and the corresponding rcaps; i.e., the rcap serves as an index into a table of rspecs.

The rspec describes a slice's privileges and resource reservations over time. Each rspec consists of a list of reservations for physical resources (e.g., CPU cycles, link bandwidth, disk capacity), limits on logical resource usage (e.g., file descriptors), assignments of shared name spaces (e.g., TCP and UDP port numbers), and other slice privileges (e.g., the right to create a virtual machine on the node). The rspec also specifies the start and end times of the interval over which these values apply.

Once acquired, rcaps can be passed from one service to another. The resources associated with the rcap are bound

4 Planetlab OS

This section defines the PlanetLab OS, the per-node software upon which the global slice abstraction is built. The PlanetLab OS consists of a Linux 2.4-series kernel with patches for vservers and hierarchical token bucket packet scheduling; the SILK (Scout in Linux Kernel) module [5, 32] that provides CPU scheduling, network accounting, and safe raw sockets; and the node manager, a trusted domain that contains slice bootstrapping machinery and node monitoring and management facilities. We describe the functionality provided by these components and discuss how it is used to implement slices, focusing on four main areas: the VM abstraction, resource allocation, controlled access to the network, and system monitoring.

4.1 Node Virtualization

A slice corresponds to a distributed set of virtual machines. Each VM, in turn, is implemented as a *vserver* [25]. The vserver mechanism is a patch to the Linux 2.4 kernel that provides multiple, independently managed virtual servers running on a single machine. Vservers are the principal mechanism in PlanetLab for providing virtualization on a single node, and contextualization of name spaces; e.g., user identifiers and files.

As well as providing security between slices sharing a node, vservers provide a limited root privilege that allows a slice to customize its VM as if it was a dedicated machine. Vservers also correspond to the resource containers used for isolation, which we discuss in section 4.2.

4.1.1 Interface

Vservers provide virtualization at the system call level by extending the non-reversible isolation provided by `chroot` for filesystems to other operating system resources, such as processes and SysV IPC. Processes within a vserver are given full access to the files, processes, SysV IPC, network interfaces, and accounts that can be named in their containing vserver, and are otherwise denied access to system-wide operating system resources. Each vserver is given its own UID/GID namespace, along with a weaker form of `root` that provides a local superuser without compromising the security of the underlying machine.

Despite having only a subset of the true superuser's capabilities, vserver `root` is still useful in practice. It can modify the vserver's root filesystem, allowing users to customize the installed software packages for their vserver. Combined with per-vserver UID/GID namespaces, it allows vservers to implement their own account management schemes (e.g., by maintaining a per-vserver `/etc/passwd` and running `sshd` on a different TCP port), thereby providing the basis for integration with other wide-area testbeds such as NetBed [42] and RON [1].

A vserver is initialized with two pieces of persistent state: a set of SSH keys and a vserver-specific `rc.vinit` file. The former allow the owners of the slice to SSH into the vserver, while the latter serves as a boot script that gets executed each time the vserver starts running.

Vservers communicate with one another via IP, and not local sockets or other IPC functions. This strong separation between slices simplifies resource management and isolation between vservers, since the interaction between two vservers is independent of their locations. However, the namespace of network addresses (IP address and port numbers) is not contextualized: this would imply either an IP address for each vserver, or hiding each vserver behind a per-node NAT. We rejected both these options in favor of slices sharing port numbers and addresses on a single node.

4.1.2 Implementation

Each vserver on a machine is assigned a unique *security context*, and each process is associated with a specific vserver through its security context. A process's security context is assigned via a new system call and inherited by the process's descendants. Isolation between vservers is enforced at the system call interface by using a combination of security context and UID/GID to check access control privileges and decide what information should be exposed to a given process. All of these mechanisms are implemented in the baseline vserver patch to the kernel. We have implemented several utilities to simplify creating and destroying vservers, and to transparently redirect a user into the vserver for his or her specific slice using SSH.

On PlanetLab, our utilities initialize a vserver by creating a mirror of a reference root filesystem inside the vserver using hard links and the "immutable" and "immutable invert" filesystem bits. Next we create two Linux accounts with login name equal to the slice name, one in the node's primary vserver and one in the vserver just created, and sharing a single UID. The default shell for the account in the main vserver is set to `/bin/vsh`, a modified `bash` shell that performs the following four actions upon login: switching to the slice's vserver security context, `chroot`ing to the vserver's root filesystem, relinquishing a subset of the true superuser's capabilities, and redirecting into the other account inside the vserver. The result of this two-account arrangement is that users accessing their virtual machines remotely via SSH/SCP are transparently redirected into the appropriate vserver and need not modify any of their existing service management scripts.

By virtualizing above a standard Linux kernel, vservers achieve substantial sharing of physical memory and disk space, with no active state needed for idle vservers. For physical memory, savings are accrued by having single copies of the kernel and daemons, and shared read-only and copy-community has not required the ability to run multiple operating systems, and so PlanetLab is able to take advantage of the efficiency of supporting a single OS API.

A slightly higher-level approach is to use *paravirtualization*, proposed by so-called isolation kernels like Xen [4] and Denali [41]. Short of full virtualization of the hardware, a subset of the processor's instruction set and some specialized virtual devices form the virtual machine exported to users. Because the virtual machine is no longer a replica of a physical machine, operating systems must be ported to the new "architecture", but this architecture can support virtualization far more efficiently. Paravirtualizing systems are not yet mature, but if they can be shown to scale, they represent a promising technology for PlanetLab.

The approach we adopted is to virtualize at the system-call level, similar to commercial offerings like Ensim [14], and projects such as User Mode Linux [12], BSD's Jail [21], and Linux vservers [25]. Such high-level virtualization adequately supports PlanetLab's goals of supporting large numbers of overlay services, while providing reasonable assurances of isolation.

3.2 Isolation and Resource Allocation

A second, orthogonal challenge is to isolate virtual machines. Operating systems with the explicit goal of isolating application performance go back at least as far as the KeyKOS system [18], which provided strict resource accounting between mutually antagonistic users. More recently, isolation mechanisms have been explored for multimedia support, where many applications require soft real-time guarantees. Here the central problem is *crosstalk*, where contention for a shared resource (often a server process) prevents the OS from correctly scheduling tasks. This has variously been addressed by sophisticated accounting across control transfers as in Processor Capacity Reserves [27], scheduling along data paths as in Scout [28], or entirely restructuring the OS to eliminate server processes in the data path as in Nemesis [23]. The PlanetLab OS borrows isolation mechanisms from Scout, but the key difference is in how these mechanisms are controlled, since each node runs multiple competing tasks that belong to a global slice, rather than a purely local set of cooperating tasks.

The problem of distributed coordination of resources, in turn, has been explored in the context of Condor [26] and more recently the Open Grid Services Architecture [16]. However, both these systems are aimed at the execution of batch computations, rather than the support of long-running network services. They also seek to define complete architectures within which such computations run. In PlanetLab the requirements are rather different: the platform must support multiple approaches to creating and binding resources to slices. To illustrate this distinction, we point out that both the Globus grid toolkit and the account management system of the Emulab testbed [42] have been implemented above PlanetLab, as have more service-oriented frameworks like SHARP [17].

3.3 Network Virtualization

Having settled on node virtualization at the system call level, the third challenge is how to virtualize the network. Vertically-structured operating systems like Exokernel and Nemesis have explored allowing access to raw network devices by using filters on send and receive [6, 13]. The PlanetLab OS uses a similar approach, providing shared network access using a "safe" version of the raw socket interface.

Exokernels have traditionally either provided the raw (physical) device to a single library OS to manage, or controlled sharing of the raw device between library OSes based on connections. However, in PlanetLab the kernel must take responsibility for sharing raw access (both reception and transmission of potentially arbitrary packets) among multiple competing services in a controlled manner according to some administrative policy. Additionally, it must protect the surrounding network from buggy and malicious services, an issue typically ignored by existing systems.

An alternative to sharing and partitioning a single network address space among all virtual machines is to *contextualize* it—that is, we could present each VM with its own local version of the space by moving the demultiplexing to another level. For instance, we could assign a different IP address to each VM and allow each to use the entire port space and manage its own routing table. The problem is that we simply do not have enough IPv4 addresses available to assign on the order of 1000 to each node.

3.4 Monitoring

A final, and somewhat new challenge is to support the monitoring and management of a large distributed infrastructure. On the network side, commercial management systems such as HP OpenView and Micromuse Netcool provide simplified interfaces to routing functionality, service provisioning, and equipment status checks. On the host management side, systems such as IBM's Tivoli and Computer Associates' UniCenter address the corresponding problems of managing large numbers of desktop and server machines in an enterprise. Both kinds of systems are aimed at single organizations, with well-defined applications and goals, seeking to manage and control the equipment they own. Managing a wide-area, evolving, federated system like PlanetLab (or the Internet as a whole) poses different challenges. Here, we are pretty much on our own.

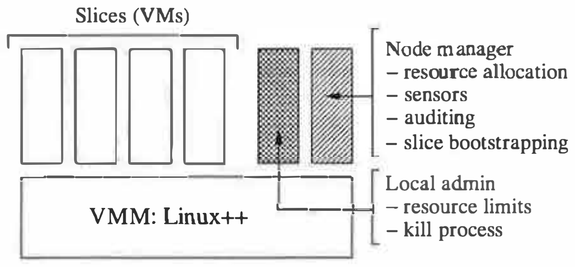


Figure 1: PlanetLab Node Architecture

cal issue of evolving the underlying OS that supports unbundled management.

Simply stated, the research community was ready to use PlanetLab the moment the first machines were deployed. Waiting for a new OS tailored for broad-coverage services was not an option, and in any case without first gaining some experience, no one could fully understand what such a system should look like. Moreover, experience with previous testbeds strongly suggested two biases of application writers: (1) they are seldom willing to port their applications to a new API, and (2) they expect a full-featured system rather than a minimalist API tuned for someone else's OS research agenda.

This suggested the strategy of starting with a full-featured OS—we elected to use Linux due to its widespread use in the research community—and incrementally transforming it based on experience. This evolution is guided by the "meta" architecture depicted in Figure 1.

At the lowest level, each PlanetLab node runs a *virtual machine monitor* (VMM) that implements and isolates VMs. The VMM also defines the API to which services are implemented. PlanetLab currently implements the VMM as a combination of the Linux kernel and a set of kernel extensions, as outlined in Section 4.

A privileged, "root" VM running on top of the VMM, called the *node manager*, monitors and manages all the VMs on the node. Generally speaking, the node manager enforces policies on creating VMs and allocating resources to them, with services interacting with the node manager to create new VMs rather than directly calling the VMM. Moreover, all interactions with the node manager are local: only services running in some other VM on the node are allowed to call the node manager, meaning that remote access to a specific node manager is always indirect through one of the services running on the node. Today, most policy is hard-coded into the node manager, but we expect that local administrators will eventually be able to configure the policies on their own nodes. (This is the purpose of the local administrator VM shown in Figure 1.)

A subset of the services (slices) running on top of the VMM can be characterized as *privileged* in some way: they are allowed to make privileged calls to the node manager (e.g., to allocate local resources to a VM). We expect all slices that provide a service to end-users to be unprivileged, while some infrastructure services may need to run in a privileged slice. To date, three types of infrastructure services are emerging: (1) brokerage services that are used to acquire resources and create slices that are bound to them, (2) environment services that are used to initialize and maintain a slice's code base, and (3) monitoring services that are used to both discover the available resources and monitor the health of running services.

Because we expect new facilities to be incorporated into the architecture over time, the key question is where any new functionality should be implemented: in an unprivileged slice, in a privileged slice, in the node manager, or in the VMM? Such decisions are guided by the following two principles:

- Each function should be implemented at the "highest" level possible, that is, running a service in a slice with limited privileged capabilities is preferred to a slice with widespread privileges, which in turn is preferred to augmenting the node manager, all of which are preferable to adding the function to the VMM.
- Privileged slices should be granted the minimal privileges necessary to support the desired behavior. They should not be granted blanket superuser privileges.

3 Design Alternatives

The PlanetLab OS is a synthesis of existing operating systems abstractions and techniques, applied to the new context of a distributed platform, and motivated by the requirements discussed in the previous section. This section discusses how PlanetLab's requirements recommend certain approaches over others, and in the process, discusses related work.

3.1 Node Virtualization

The first challenge of the PlanetLab OS is to provide a virtual machine abstraction for slices; the question is, at what level? At one end of the spectrum, full hypervisors like VMware completely virtualize the physical hardware and thus support multiple, unmodified operating system binaries. If PlanetLab were to supply this low level of virtualization, each slice could run its own copy of an OS and have access to all of the devices and resources made available to it by the hypervisor. This would allow PlanetLab to support OS kernel research, as well as provide stronger isolation by removing contention for OS resources. The cost of this approach is performance: VMware cannot support the number of simultaneous slices required by PlanetLab due to the large amount of memory consumed by each machine image. Thus far, the PlanetLab to slices after the fact. This concern about how users (or their services) affect the outside world is a novel requirement for PlanetLab, unlike traditional timesharing systems, where the interactions between users and unsuspecting outside entities is inherently rare.

Security was recognized from the start as a critical issue in the design of PlanetLab. However, effectively limiting and auditing legitimate users has turned out to be just as significant an issue as securing the OS to prevent malicious users from hijacking machines. For example, a single PlanetLab user running TCP throughput experiments on U.C. Berkeley nodes managed to consume over half of the available bandwidth on the campus gateway over a span of days. Also, many experiments (e.g., Internet mapping) have triggered IDS mechanisms, resulting in complaints that have caused local administrators to pull the plug on nodes. The Internet has turned out to be unexpectedly sensitive to the kinds of traffic that experimental planetary-scale services tend to generate.

2.2 Unbundled Management

Planetary-scale services are a relatively recent and ongoing subject of research; in particular, this includes the services required to manage a global platform such as PlanetLab. Moreover, it is an explicit goal of PlanetLab to allow independent organizations (in this case, research groups) to deploy alternative services in parallel, allowing users to pick which ones to use. This applies to application-level services targeted at end-users, as well as *infrastructure services* used to manage and control PlanetLab itself (e.g., slice creation, resource and topology discovery, performance monitoring, and software distribution). The key to unbundled management is to allow parallel infrastructure services to run in their own slices of PlanetLab and evolve over time.

This is a new twist on the traditional problem of how to evolve a system, where one generally wants to try a new version of some service in parallel with an existing version, and roll back and forth between the two versions. In our case, multiple competing services are simultaneously evolving. The desire to support unbundled management leads to two requirements for the PlanetLab OS.

- To minimize the functionality subsumed by the PlanetLab OS—and maximize the functionality running as services on top of the OS—*only local (per-node) abstractions* should be directly supported by the OS, allowing all global (network-wide) abstractions to be implemented by infrastructure services.

- To maximize the opportunity for services to compete with each other on a level playing field, the interface between the OS and these infrastructure services must be *sharable*, and hence, without special privilege. In other words, rather than have a single privileged application controlling a particular aspect of the OS, the PlanetLab OS potentially supports many such management services. One implication of this interface being sharable is that it must be well-defined, explicitly exposing the state of the underlying OS. In contrast, the interface between an OS and a privileged control program running in user space is often ad hoc since the control program is, in effect, an extension of the OS that happens to run in user space.

Of particular note, *slice creation* is itself implemented as a service running in its own slice, which leads to the following additional requirement on the PlanetLab OS:

- It must provide a *low-level interface for creating a VM* that can be shared by multiple slice creation services. It must also host a "bootstrapping" slice creation service to create initial slices, including the slices that other slice creation services run in.

An important technical issue that will influence how the slice abstraction evolves is how quickly a network-wide slice can be instantiated. Applications like the ones listed in the Introduction are relatively long-lived (although possibly modified and restarted frequently), and hence the process of creating the slice in which they run can be a heavy-weight operation. On the other hand, a facility for rapidly establishing and tearing down a slice (analogous to creating/destroying a network connection) would lead to slices that are relatively short-lived, for example, a slice that corresponds to a communication session with a known set of participants. We evaluate the performance of the current slice creation mechanism in Section 5. It is not yet clear what other slice creation services the user community will provide, or how they will utilize the capability to create and destroy slices.

The bottom line is that OS design often faces a tension between implementing functionality in the kernel and running it in user space, the objective often being to minimize kernel code. Like many VMM architectures, the PlanetLab OS faces an additional, but analogous, tension between what can run in a slice or VM, and functionality (such as slice user authentication) that requires extra privilege or access but is not part of the kernel. In addition, there is a third aspect to the problem that is peculiar to PlanetLab: functionality that can be implemented by parallel, competing subsystems, versus mechanisms which by their very nature can only be implemented once (such as bootstrapping slice creation). The PlanetLab OS strives to minimize the latter, but there remains a core of non-kernel functionality that has to be unique on a node.

2.3 Evolving Architecture

While unbundled management addresses the challenge of evolving PlanetLab as a whole, there remains the very practiedgment indicates that the write will ultimately be seen by all or none of the replicas. A user may choose to re-submit an un-acknowledged write, and Om performs appropriate duplicate detection and elimination.

After a failure-induced reconfiguration and before a new primary can serialize any new writes, it first collects all pending writes from the replicas in the new configuration and processes the writes again using the normal two-phase protocol. Each replica performs appropriate duplicate detection and elimination in this process. Such design solves the previous problem because if any read sees a write, then the write must be either applied or in the pending queue on all replicas.

4 Reconfiguration

Each configuration has a monotonically increasing sequence number, increased with every reconfiguration. For any configuration and at any point of time, a replica can only be in a single reconfiguration process (either failure-free or failure-induced). It is however, possible that different replicas in the same configuration are simultaneously in different reconfiguration processes.

Conceptually, a replica that finishes reconfiguration will try to inform other replicas of the new configuration by sending *configuration notices*. In failure-free reconfigurations, only the primary does this, because the other replicas are passive. In failure-induced reconfigurations, all replicas transmit configuration notices to aid in completing reconfiguration earlier. In many cases, most replicas do not even need to enter the consensus protocol—they simply wait for the configuration notice (within a timeout).

4.1 Failure-free Reconfiguration

Only the primary may initiate failure-free reconfiguration. Secondary replicas are involved only when i) the primary transmits to them data for creating new replicas; and ii) the primary transmits configuration notices.

The basic mechanism of failure-free reconfiguration is straightforward. After transferring data to the new replicas in two stages (snapshot followed by logged writes as discussed earlier), the primary constructs a configuration for the new desired membership. This new configuration will have a new `sequenceNum` by incrementing the old `sequenceNum`. The `consensusID` of the configuration remains unchanged.

The primary then informs the other replicas of the new configuration and waits for acknowledgments. If timeout occurs, a failure-induced reconfiguration will follow.

```
// A snapshot of the current configuration must be passed in.
void shrink(Configuration dupconf)
    throws InterruptedException {
    //Stop granting leases for current configuration.
    current_configuration.valid = false;

    newmember = set of replicas I can reach in dupconf;
    newconf = new Configuration(newmember);
    newconf.sequenceNum = dupconf.sequenceNum + 2;
    newconf.consensusID = dupconf.name + "_" +
        newconf.sequenceNum;

    decision = consensus(newconf, dupconf.consensusID);
    leaseManager.waitForLeaseExpire(dupconf);

    Block writes and configuration notices;
    if (current_configuration.sequenceNum <
        decision.sequenceNum) {
        current_configuration = decision;
        send configuration notices;
        if (I am primary in decision) applyPendingWrites();
    }
    // If not, then configuration notice received.
    // dupconf is no longer current and reconfig is obsolete.
    Unblock writes and configuration notices;
}
```

Figure 2: Failure-induced reconfiguration.

4.2 Failure-induced Reconfiguration

In contrast to failure-free reconfigurations, failure-induced reconfigurations can only shrink the replica group (potentially followed by failure-free reconfigurations to expand the replica group as necessary). Doing this simplifies design because failure-induced reconfigurations do not need to create new replicas and request them to participate in the consensus protocol. Failure-induced reconfigurations can take place during normal operations, failure-free reconfigurations or even failure-induced reconfigurations.

A replica initiates failure-induced reconfiguration (Figure 2) upon detecting a failure. The replica first disables the current configuration so that leases can no longer be granted for the current configuration. This reduces the time we need to wait for lease expiration later. Next, it will perform another round of failure detection for all members of the configuration. The result (a subset of the current replicas) will be used as a *proposal* for the new configuration. The replica then invokes a consensus protocol, which returns a *decision* that is agreed upon by all replicas entering the protocol. When invoking the consensus protocol, the replica needs to pass a unique ID for this particular invocation of the consensus protocol. Otherwise, since nodes can be arbitrarily slow, different invocations of the consensus protocol may interfere with one another.

Before adopting a decision, each replica needs to wait for all leases to expire with respect to the old configuration. Finally, the primary of the new configuration will collect and re-apply any pending writes. When re-applying pending writes, the primary only waits for a certain timeout. If a subsequent failure were to take place, the replicas will start another failure-induced reconfiguration.

One important optimization to the previous protocol is that after a replica determines `newmember`, it checks whether it has the smallest ID in the set. If it does not, the replica will wait (within a timeout) for a configuration notice. With this optimization, in most cases, only a single replica enters the consensus protocol, which can significantly improve the time complexity of the randomized consensus protocol (see Section 5.3).

When a failure-induced reconfiguration is invoked in the middle of a failure-free reconfiguration, they may interfere with each other and result in inconsistency. Such issue is properly addressed in our complete design [42].

5 Single Replica Regeneration

Failure-induced reconfigurations depend on a consensus protocol to ensure the uniqueness of the new configuration and in turn, data consistency. Consensus [22] is a classic distributed computing problem and we can conceptually use any consensus protocol in Om. However, most consensus protocols such as Paxos [21] rely on majority quorums and thus cannot tolerate more than $n/2$ failures among n replicas. To reduce the number of replicas required to carry out regeneration (as a desirable side-effect, this also reduces the overhead of acquiring leases and of performing writes), we adopt the *witness model* [40] to achieve probabilistic consensus without requiring a majority.

5.1 Probabilistic Quorum Intersection without Majority

The witness model [40] is a novel quorum design that allows quorums to be as small as a single node, while ensuring probabilistic quorum intersection. In our system, for each new configuration, the primary chooses $m \times t$ witnesses and communicates their identities to all secondary replicas. Witnesses are periodically probed by the primary and refreshed as necessary upon failure. This refresh is trivial and can be done in the form of a two-phase write. If failure occurs between the first and the second phase, a replica will use both old and new witnesses in the consensus protocol. The primary may utilize a variety of techniques to choose witnesses, with the goal of choosing witnesses with small failure correlation and diversity in the set of network paths from the replicas to individual witnesses. For example, the primary may simply use entries from its finger table under Chord [38].

For now, we will consider replicas that are not in *singleton partitions*, where a single node, LAN, or perhaps a small autonomous system is unable to communicate with the rest of the network. Later we will discuss how to determine singleton partitions. We say that a replica can *reach* a witness if a reply can be obtained from the witness within a certain timeout. The witness model utilizes the following *limited view divergence* property:

Consider a set S of functioning randomly-placed witnesses that are not co-located with the replicas (e.g., not in the same LAN). Suppose one replica A can reach the subset S_1 of witnesses and cannot reach the subset S_2 of witnesses (where $S_1 \cup S_2 = S$). Then the probability that another replica B cannot reach any witness in S_1 and can reach all witnesses in S_2 decreases with increasing size of S.

Intuitively, the property says that two replicas are unlikely to have a completely different view regarding the reachability of a set of randomly-placed witnesses. The size of S and the resulting probability are thoroughly studied in [40] using the RON [4] and TACT [41] traces. Later we will also present additional results based on PlanetLab measurements.

The validity of limited view divergence can probably be explained by the rarity [9] of large-scale "hard partitions", where a significant fraction of Internet nodes are unable to communicate with the rest of the network. Given that witnesses are randomly placed, if the two replicas have completely different views on the witnesses, this tends to indicate a "hard partition". Further, the more witnesses, the larger-scale the partition would have to be to result in entirely disjoint views from the perspective of two independent replicas.

To utilize the limited view divergence property, all replicas logically organize the witnesses into an $m \times t$ matrix. The number of rows, m, determines the probability of intersection. The number of columns, t, protects

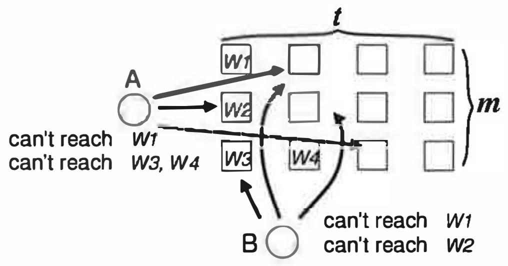


Figure 3: Two replicas and 3 × 4 witnesses.

against the failure of individual witnesses, so that each row has at least one functioning witness with high probability. Each replica tries to coordinate with one witness from each row. Specifically, a replica uses the first witness from left to right that it can reach for each row (Figure 3). The set of witnesses used by a replica is its quorum. Now consider two replicas A and B. The desirable outcome is that A's quorum intersects with B's. It can be shown that if the two quorums do not intersect, with high probability (in terms of t), A and B have completely different views on the reachability of m witnesses [40].

Replicas behind singleton partitions will violate limited view divergence. However, if the witnesses are not colocated with the replica, then the replica behind the partition will likely not be able to reach *any* witness. As a result, it cannot acquire a quorum and will thus block. This is a desirable outcome as the replicas on the other side of the partition will reach consensus on a new configuration that excludes the node behind the singleton partition. To better detect singleton partitions, a replica may also check whether all reachable witnesses are within its own LAN or autonomous system.

5.2 Emulating Probabilistic Shared-Memory

We intend to substitute the majority quorum in traditional consensus protocols with the witness model, so that the consensus protocol can achieve probabilistic consensus without requiring majority. To do this however, we need a consensus protocol with "good" termination properties for the following reason. Non-intersection in the witness model is ultimately translated into the *unsafety* (probability of having multiple decisions) of a consensus protocol. Unsafety in turn, means violation of consistency in Om. For protocols with multiple rounds, unsafety potentially increases with every round. This precludes the application of protocols such as Paxos [21] that do not have good termination guarantees.

To address the previous issue, we first use the witness model to emulate a *probabilistic shared-memory*, where reads may return stale values with a small probability. We then apply a shared-memory randomized consensus protocol [36], where the expected number of rounds before termination is constant and thus helps to bound unsafety.

To reduce the message complexity of the shared-memory emulation, we choose not to directly emulate [40] the standard notion of reads and writes. Rather, we define an *access* operation on the shared-memory to be an update to an array element followed by a read of the entire array. The element to be updated is indexed by the replica's identifier. The witnesses maintain the array. Upon receiving an access request, a witness updates the corresponding array element and returns the entire array. Such processing is performed in isolation from other access requests on the same witness. Figure 4 provides the pseudo-code for such emulation.

```
For Replicas:
static int version = 0;
int[] access(String arrayname, int newvalue) {
    Record[][] replies = new Record[m][];
    replies[1..m] = null; version++; j = 1;
    while ((∃ i, replies[i] == null) and ( j ≤ t)) {
        send (myindex, version, newvalue) to all witness[i][j]
            where (replies[i] == null);
        wait until all replies received or time out;
        replies[i] = the reply from witness[i][j];
        j++;
    }

    if (replies[1..m] == null) block;
    int[] result = new int[n]; // combine all replies
    for (int k = 1; k ≤ n; k++)
        result[k] = replies[i][k].value, where replies[i][k]
            has the largest version in replies[1..m][k]
    return result;
}

For Witnesses:
Record[] processAccess(String arrayname, int index,
                        int version, int newvalue) {
    let record[1..n] be the array corresponding to arrayname;
    if (record[index].version < version) {
        record[index].version = version;
        record[index].value = newvalue;
    }
    return record;
}
```

Figure 4: Emulating shared-memory under the witness model.

```
// Shared data: The $i$th iteration uses two arrays,
// proposed[i] and check[i]. Each array has $n$ entries,
// one for each replica. All entries initialized to null.
int randCons(int proposal) {
    i = 0; myvalue = proposal;
    while (true) {
        i++;
        prop_view = access(proposed[i], myvalue);
        if (different proposals appear in prop_view)
            check_view = access(check[i], 'disagree');
        else
            check_view = access(check[i], 'agree');

        if (check_view only contains 'agree')
            return myvalue; //this is the decision
        if (check_view only contains 'disagree')
            myvalue = a random element in prop_view
                indexed by coinFlip();
        if (check_view has both 'agree' and 'disagree')
            myvalue = prop_view[q],
                $\forall$ q, where check_view[q] == 'agree';
    }
}
```

Figure 5: Randomized consensus protocol for shared-memory.

While the access primitive appears to be a simple wrapper around reads and writes, it actually violates atomicity and qualitatively changes the semantics of the shared-memory. It reduces the message (and time) complexity of the shared-memory emulation in [40] by half. More details are available in [42].

5.3 Application of Shared-memory Randomized Consensus Protocol

With the shared-memory abstraction, we can now apply a previous shared-memory consensus protocol [36] (Figure 5). For simplicity, we assume that the proposals and decisions are all integer values, though they are actually configurations. In the figure, we already substitute the read and write operations in the original protocol with our new access operations. We implement **coinFlip**() using a local random number generator initialized using a common seed shared by all replicas. Such implementation is different and simpler than the design for standard shared-memory consensus protocols, and it reduces the complexity of the protocol by a factor of $\theta(n^2)$. See [42] for details on why such optimization is possible.

The intuition behind the shared-memory consensus protocol is subtle and several textbooks have chapters devoted to these protocols (e.g., Chapter 11.4 of [8]). Since the protocol itself is not a contribution of this paper, we only enumerate several important properties of the protocol. Proofs are available in [42].

- The protocol proceeds in successive iterations, each iteration has two accesses. Each access requires one round of communication (between the replicas and the witnesses), and needs to coordinate with a quorum. Non-intersection for any access may result in unsafety.
- Each iteration has a certain probability of terminating. The number of iterations before termination is a random variable.
- With two distinct proposals, the expected time complexity of the protocol is below 3.1 iterations (6.2 rounds).
- If all replicas entering the protocol have the same proposal (or if only one replica enters the protocol), the protocol terminates (deterministically) after one iteration. With the optimization in Section 4.2, this will be the situation when the new primary does not crash in the middle of reconfiguration.

6 Experimental Evaluation

This section evaluates the performance and unsafety of Om. Availability of Om and the benefit of single replica regeneration is studied separately [42]. Om is written in Java 1.4, using TCP and nonblocking I/O for communication. All messages are first serialized using Java serialization and then sent via TCP. The core of Om uses an event-driven architecture.

6.1 Unsafety Evaluation

Om is able to regenerate from any single replica at the cost of a small probability of consistency violation. We first quantify such unsafety under typical Internet conditions.

Unsafety is about rare events, and explicitly measuring unsafety experimentally faces many of the same challenges as evaluating service availability [41]. For instance, assuming that each experiment takes 10 seconds to complete, we would need on average over four years to observe a single inconsistency event for an unsafety of 10^{-7}. Given these challenges, we follow the methodology in [41] and use a real-time emulation environment for our evaluation. We instrument Om to add an artificial delay to each message. Since the emulation is performed on a LAN, the actual propagation delay is negligible. We determine the distribution of appropriate artificial delays by performing a large-scale measurement study of PlanetLab sites. For our emulation, we set the delay of each message sent across the LAN to the delay of the corresponding message in our WAN measurements.

Our WAN sampling software runs with the same communication pattern as the consensus protocol except that it does not interpret the messages. Rather, the replicas repeatedly communicate with all witnesses in parallel via TCP. The request size is 1KB while the reply is 2KB. We log the time (with a cap of 6 minutes) needed to receive a reply from individual witnesses. The sampling interval (time between successive samples) for each replica ranges from 1 to 10 seconds in different measurements. Notice that we do not necessarily wait for the previous probe's reply before sending the next probe. All of our measurements use 7 witnesses and 15 replicas on 22 different PlanetLab sites. To avoid the effects of Internet2 and to focus on the pessimistic behavior of less well-connected sites, we locate the witnesses at non-educational or foreign sites: Intel Research Berkeley, Technische Universitat Berlin, NEC Laboratories, Univ of Technology, Sydney, Copenhagen, ISI, Princeton DSL. Half of the nodes serving as replicas are also foreign or non-educational sites, while the other half are U.S. educational sites. For the results presented in this paper, we use an 8-day long trace measured in July 2003. The sampling interval in this trace is 5 seconds, and the trace contains $150,000$ intervals. Each interval has $7 \times 15 = 105$ samples, resulting in over 15 million samples.

6.2 Unsafety Results

The key property utilized by the witness model is that P_{ni} (probability of non-intersection) can be quite small even with a small number of witnesses. Earlier work [40] verifies this assumption using a number of existing network measurement traces [4, 41]. In the RON1 trace, 5 witness rows result in 4×10^{-5} P_{ni}, while it takes 6 witness rows to yield similar P_{ni} under the TACT trace.

Given these earlier results, this section concentrates on the relationship between P_{ni} and unsafety, namely, how the randomized consensus protocol amplifies P_{ni} into unsafety under different parameter settings. This is important since the protocol has multiple rounds, and non-intersection in any round may result in unsafety.

Unsafety can be affected by several parameters in our system: the message timeout value for contacting witnesses, the size of the witness matrix and the number of

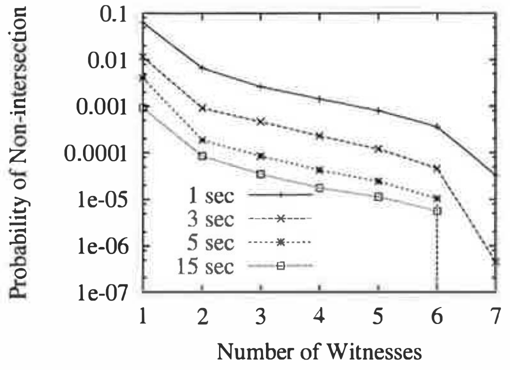


Figure 6: P_{ni} for different time-out values.

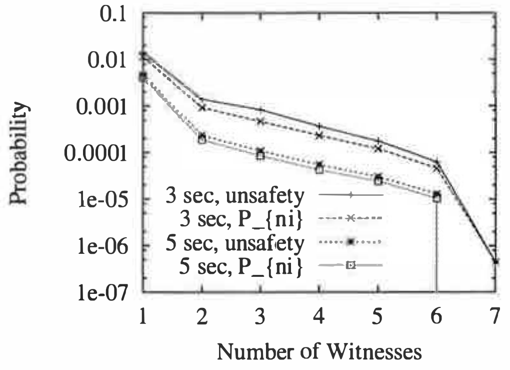


Figure 7: Unsafety and P_{ni}.

replicas. Since a larger t value in the witness matrix is used to guard against potential witness failures and witnesses do not fail in our experiments, we use $t = 1$ for all our experiments. Witness failures between accesses may slightly increase P_{ni}, but a simple analysis can show that such effects are negligible [42] under practical parameters. Larger timeout values decrease the possibility that a replica cannot reach a functioning witness and thus decrease P_{ni}. Figure 6 plots P_{ni} for different timeout values. In our finite-duration experiments, we cannot observe probabilities below 10^{-7}. This is why the curves for 5 and 15 second timeout values drop to zero with seven witnesses. The figure shows that P_{ni} quickly approaches its lowest value with the timeout at 5 seconds.

Having determined the timeout value, we now use emulation to measure unsafety. We first consider the simple case of two replicas. Figure 7 plots both P_{ni} and unsafety for two different timeout values. Using just 7 witnesses, Om already achieves an unsafety of 5×10^{-7}. With 5 replicas and a pessimistic replica MTTF of 12 hours, reconfiguration takes place every 2.4 hours. With unsafety at 5×10^{-7}, an inconsistent reconfiguration would take place once every 500 years. In a peer-to-

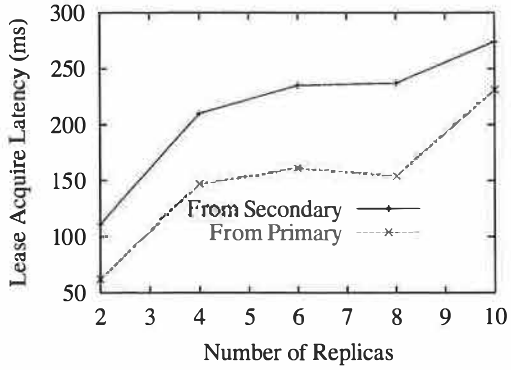


Figure 8: Latency for renewing leases based on our lease graph.

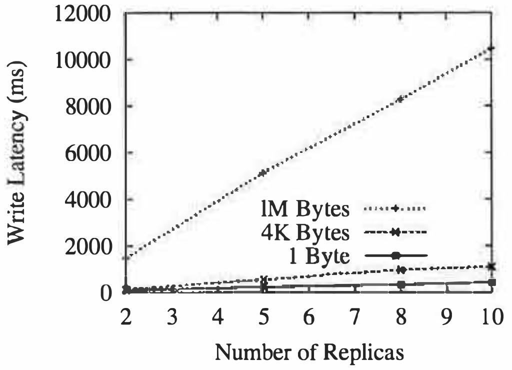


Figure 9: Latency for a write.

peer system with a large number of nodes, reconfiguration can occur much more frequently. For example, for a Pastry ring with $1,000$ nodes and replication degree of 5, each node may be shared by 5 different configurations. As a result, reconfiguration in the entire system occurs every 8.64 seconds. In this case, inconsistent regeneration will take place once every half year system-wide. It may be possible to further reduce unsafety with additional witnesses, though the benefits cannot be quantified with the granularity of our current measurements.

The extended version of this paper [42] further discusses the relationship between unsafety and P_{ni}, and also generalizes the results to more than two replicas. Due to space limitations, we will move on to the performance results.

6.3 Performance Evaluation

We obtain our performance results by deploying and evaluating Om over PlanetLab. In all our performance experiments, we use the seven witnesses used before in our WAN measurement. With single replica regeneration, Om can achieve high availability with a small number of replicas. For example, our analysis [42] shows that Om can achieve 99.9999% availability with just 4 replicas under reasonable parameter settings. Thus, we focus on small replication factors in our evaluation.

6.3.1 Normal Case Operations

We first provide basic latency results for individual read and write operations using 10 PlanetLab nodes as replicas. We intentionally choose a mixture of US educational sites, US non-educational sites and foreign sites. To isolate the performance of Om from that of Pastry, we inject reads and writes from the replicas, instead of having client nodes injecting accesses via peer-to-peer routing.

Since a read in Om is processed by a single replica (as long as it holds all necessary leases), a read involves only a single request/response pair. However, additional latency is incurred when lease renewal is required. To separate these effects, we directly study the latency of lease renewal. However, notice that though not implemented in our prototype, leases can be renewed proactively, which will hide most of this latency from the critical path. Figure 8 plots the time needed to renew leases based on our lease graph. Obviously, the primary incurs smaller latency to renew all of its leases. Secondary replicas need to contact the primary first to request the appropriate set of subleases.

Processing writes is more complex because it involves a two-phase protocol among the replicas. Figure 9 presents the latency for writes of different sizes. In all three cases, the latency increases linearly with the number of replicas, indicating that the network bandwidth of the primary is the likely bottleneck for these experiments. For 1MB writes, the latency reaches 10 seconds for 10 replicas. We believe such latency can be improved by constructing an application-layer multicast tree among the replicas.

6.3.2 Reconfiguration

We next study the performance of regeneration. For these experiments, we use five PlanetLab nodes as replicas: `bu.edu`, `cs.duke.edu`, `hpl.hp.com`, `cs.arizona.edu` and `cs-ipv6.lancs.ac.uk`. Figure 10 shows the cost of failure-free reconfiguration. In all cases, the two components of "finding replica set"

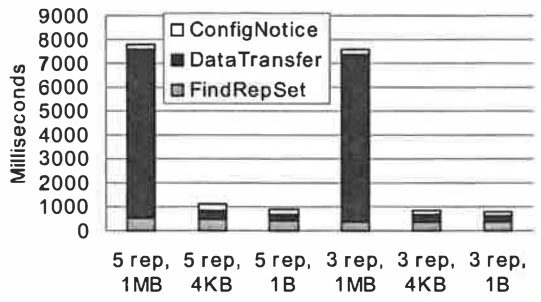


Figure 10: The cost of creating new replicas and invoking a failure-free reconfiguration. All experiments start from a single replica with a data object of a particular size and then expand to either 3 or 5 replicas.

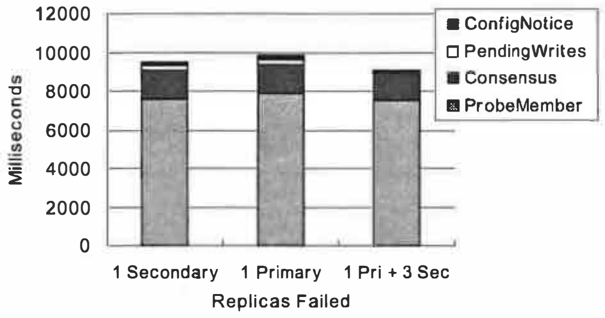


Figure 11: The cost of failure-induced reconfigurations as observed by the primary of the *new* configuration. All experiments start from a five-replica configuration and then we kill a particular set of replicas.

and "sending configuration notices" take less than one second. This is also the cost of failure-free reconfigurations when we shrink instead of expand the replica group. The latency of "finding replica set" is determined by Pastry routing, the only place where Pastry's performance influences the performance of reconfiguration. The time needed to transfer the data object begins to dominate the overall cost with 1MB of data. We thus believe that new replicas should be regenerated in the background using bandwidth consumption controlling techniques such as TCP Nice [39].

The cost of failure-induced reconfiguration is higher. Figure 11 plots the cost of failure-induced reconfiguration as observed by the primary of the *new* configuration. Using optimizations in Section 4.2, only one replica (the one with the smallest ID, which is also the primary of the new configuration) enters the consensus protocol immediately, while other replicas wait for a timeout (10 seconds in our case). As a result of this optimization, in all three cases, the consensus protocol terminates after one iteration (two rounds) and incurs an overhead of roughly 1.5 seconds. The new primary then notifies the other replicas of the resulting configuration. In Figure 11, the time needed to determine the live members of the old configuration dominates the total overhead. This step involves probing the old members and waiting for replies within a timeout (7.5 seconds in our case). A smaller timeout would decrease the delay, but would also increase the possibility of false failure detection and unnecessary replica removal.

Waiting for lease expiration, interestingly, does not cause any delay in our experiments (and thus is not shown in Figure 11). Since we disable lease renewal at the very beginning of the protocol and our lease duration is 15 seconds, by the time the protocol completes the probing phase and the consensus protocol, all leases have already expired. In these experiments, we do not inject writes. Thus, the time for applying pending writes only includes the time for the new primary to collect pending writes from the replicas and then to realize that the set is empty. The presence of pending writes will increase the cost of this step, as explored in our later experiments.

6.3.3 End-to-end Performance

Our final set of experiments study the end-to-end effects of reconfiguration on users. For this purpose, we deploy a 42-node Pastry ring on 42 PlanetLab sites, and then measure the write throughput and latency for a particular object during reconfiguration.

For these experiments, we configure the system to maintain a replication degree of four. To isolate the throughput of our system from the potential bottleneck on a particular network path, we directly inject writes on the primary. Both the writes and the data object are of 80KB size. In the two-phase protocol for writes, the primary sends a total of 240KB data to disseminate each write to the three secondary replicas. For each write, the primary also incurs roughly 9KB of control message overhead.

The experiment records the total number of writes returned for every 5 second interval, and then reports the average as the system throughput. Our test program also records the latency experienced by each write. Writes are rejected when the system is performing a failure-induced reconfiguration.

For our experiment, we first replicate the data object at `cs.caltech.edu`, `cs.ucla.edu`, `inria.fr` and `csres.utexas.edu` (primary). Notice that this replica set is determined by Pastry. Next we manually kill the process running on `inria.fr`, thus causing a

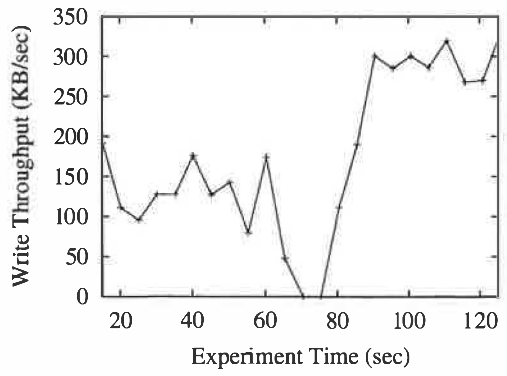


Figure 12: Measured write throughput under regeneration.

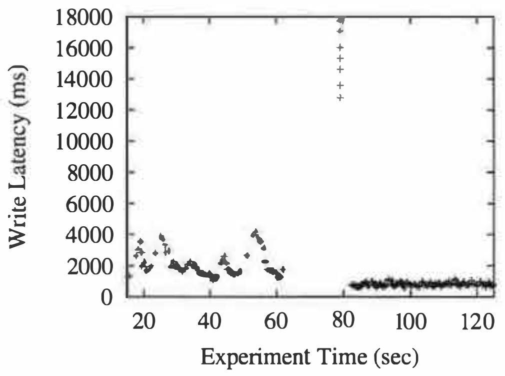


Figure 13: Measured latency of writes. For each write submitted at time t_1 and returning at time t_2, we plot a point $(t_2, t_2 - t_1)$ in the graph.

failure-induced reconfiguration to shrink the configuration to three replicas. Next, to maintain a replication factor of 4, Om expands the configuration to include `lbl.gov`.

Figure 12 plots the measured throughout of the system over time. The absolute throughput in Figure 12 is largely determined by the available bandwidth among the replica sites. The jagged curve is partly caused by the short window (5 seconds) we use to compute throughput. We use a small window so that we can capture relatively short reconfiguration activity. We manually remove `inria.fr` at $t = 62$.

The throughput between $t = 60$ and $t = 85$ in Figure 12 shows the effects of regeneration. Because of the failure at $t = 62$, the system is not able to properly process writes accepted shortly after this point. The system begins regeneration when the failure is detected at $t = 69$. The failure-induced reconfiguration shrinking the configuration takes 13 seconds, of which 3.7 is consumed by the application of pending writes. The failure-free reconfiguration that expands the configuration to include `lbl.gov` takes 1.3 seconds. After the reconfiguration, the throughput gradually increases to its maximum level as the two-phase pipeline for writes fills.

To better understand these results, we plot per-write latency in Figure 13. The gap between $t = 62$ and $t = 82$ is caused by system regeneration when the system cannot process writes (from $t = 62$ to $t = 69$) or rejects writes (from $t = 69$ to $t = 82$). At $t = 80$, those seven writes submitted between $t = 62$ and $t = 69$ return with relatively high latency. These writes have been applied as pending writes in the new configuration.

We also perform additional experiments showing similar results when regenerating three replicas instead of one replica. Overall, we believe that regenerating in 20 seconds can be highly effective for a broad array of services. This overhead can be further reduced by combining the failure detection phase (7 seconds) with the "ProbeMember" phase in failure-induced reconfiguration, potentially reducing the overhead to 13 seconds.

7 Related Work

RAMBO [15, 23] explicitly aims to support reconfigurable quorums, and thus shares the same basic goal as Om. In RAMBO, configuration not only refers to a particular set of replicas, but also includes specific quorum definitions used in accessing the replicas. In our system, the default scheme for data accessing is read-one/write-all. RAMBO also uses a consensus protocol (Paxos [21]) to uniquely determining the next configuration. Relative to RAMBO, our design has the following features. First, RAMBO only performs failure-induced reconfigurations. Second, RAMBO requires a majority of replicas to reconfigure. On the other hand, Om can reconfigure from any single replica at the cost of a small probability of violating consistency. Finally, in RAMBO, both reads and writes proceed in two phases. The first phase uses read quorums to obtain the latest version number (and value, in the case of reads), while the second phase uses a write quorum to confirm the value. Thus, reads in RAMBO are much more expensive than ours. Om avoids this overhead for reads by using a two-phase protocol for write propagation.

A unique feature of RAMBO is that it allows accesses even during reconfiguration. However, to achieve this, RAMBO requires reads or writes to acquire appropriate quorums from all previous configurations that have not been garbage-collected. To garbage-collect a configuration, a replica needs to acquire both a read and a write quorum of that configuration. This means that whenever a *read* quorum of replicas fail, the configuration can never be garbage-collected. Since both reads and writes in RAMBO need to acquire a write quorum, this further implies that RAMBO *completely* blocks whenever it loses a *read* quorum. Om uses lease graphs to avoid acquiring quorums for garbage-collection. If Om uses the same read/write quorums as in RAMBO, Om will regenerate (and thus temporarily block accesses) *only* if RAMBO blocks.

Related to replica group management, there has been extensive study on group communication [3, 5, 19, 24, 28, 29, 31] in asynchronous systems. A comprehensive survey [7] is available in this space. Group communication does not support read operations, and thus does not need leases or a two-phase write protocol. On the other hand, Om does not deliver membership views and does not require view synchrony. The membership in the configuration can not be considered as a view, since we do not impose virtual synchrony relationship between the configurations and writes.

The group membership design in [31] uses ideas similar to failure-free reconfiguration (called *update*) and failure-induced reconfiguration (called *reconfiguration*). However, updates in [31] involve two phases rather than a single phase in our failure-free reconfiguration. In fact, their updates are similar to Om writes. Furthermore, the reconfiguration process in [31] involves re-applying pending "updates". Our design avoids this overhead by using appropriate manipulation [42] on the sequence numbers proposed by failure-free and failure-induced reconfigurations.

In standard replicated state machine techniques [37], all writes go through a consensus protocol and all reads contact a read quorum of replicas. With a fixed set of replicas, a read quorum here usually cannot be a single replica. Otherwise the failure of any replica will disable the write quorum. In comparison, with regeneration functionality and the lease graph, Om is able to use a small read quorum (i.e., a single replica). Om also uses a simpler two-phase write protocol in place of a consensus protocol for normal writes. Consensus is only used for reconfiguration.

Similar to the witness model, voting with witnesses [26] allows the system to compose a quorum with nodes other than the replicas themselves. However, voting with witnesses still uses the simple majority quorum technique and thus always requires a majority to proceed. The same is true for Disk Paxos [14] where a majority of disks is needed.

8 Conclusions

Motivated by the need for consistent replica regeneration, this paper presents Om, the first read/write peer-to-peer wide-area storage system that achieves high availability and manageability through online automatic regeneration while still preserving consistency guarantees. We achieve these properties through the following three novel techniques: i) single replica regeneration that enables Om to achieve high availability with a small number of replicas; ii) failure-free reconfigurations allowing common-case reconfigurations to proceed within a single round of communication; and iii) a lease graph and two-phase write protocol to avoid expensive consensus for normal writes and also to allow reads to be processed by any replica. Experiments on PlanetLab show that consistent regeneration in Om completes in approximately 20 seconds, with the potential for further improvement to 13 seconds.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Miguel Castro, for their detailed and helpful comments, which significantly improved this paper.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[2] Akamai Corporation, 1999. `http://www.akamai.com`.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Subsystem for High Availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, July 1992.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5:47–76, February 1987.

[6] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33:1–43, December 2001.

[8] R. Chow and T. Johnson. *Distributed Operating Systems & Algorithms.* Addison Wesley Longman, Inc., 1998.

[9] R. Cohen, K. Erez, D. ben Avraham, and S. Havlin. Resilience of the Internet to Random Breakdowns. *Physical Review Letters*, 85(21), November 2000.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[12] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[13] FreePastry. `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry`.

[14] E. Gafni and L. Lamport. Disk Paxos. In *Proceedings of the International Symposium on Distributed Computing*, pages 330–344, 2000.

[15] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2003.

[16] M. K. Goldberg. The Diameter of a Strongly Connected Graph (Russian). *Doklady*, 170(4), 1966.

[17] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[18] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[19] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*, November 2000.

[21] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16:133–169, May 1998.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1997.

[23] N. Lynch and A. Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

[24] S. Mishra, L. Peterson, and R. Schlichting. Consul: A Communication Substrate for Fault-tolerant Distributed Programs. *Distributed Systems Engineering*, 1:87–103, December 1993.

[25] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[26] J.-F. Paris. Voting with Witnesses: A Consistency Scheme for Replicated Files. In *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 606–612, 1986.

[27] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet . In *Proceedings of the ACM HotNets-I Workshop*, 2002.

[28] R. D. Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A Dynamic Primary Configuration Group Communication Service. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, September 1999.

[29] R. Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. *K.P. Birman and R. van Renesse, editors, Reliable Distributed Computing with the Isis Toolkit*, pages 133–147, September 1993.

[30] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 2003.

[31] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proceedings of the 10th ACM Symposium of Principles of Distributed Computing*, pages 341–352, 1991.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[33] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[34] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[35] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[36] M. Saks, N. Shavit, and H. Woll. Optimal Time Randomized Consensus – Making Resilient Algorithms Fast in Practice. In *Proceedings of the Second Symposium on Discrete Algorithms*, pages 351–362, January 1991.

[37] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, pages 299–319, December 1990.

[38] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001*, pages 149–160, August 2001.

[39] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[40] H. Yu. Overcoming the Majority Barrier in Large-Scale Systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, October 2003.

[41] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[42] H. Yu and A. Vahdat. Consistent and Automatic Replica Regeneration. Technical Report IRP-TR-04-01, Intel Research Pittsburgh, 2004. Also available at `http://www.intel-research.net/pittsburgh/publications.asp`.

Total Recall: System Support for Automated Availability Management

Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Availability is a storage system property that is both highly desired and yet minimally engineered. While many systems provide mechanisms to improve availability – such as redundancy and failure recovery – how to best configure these mechanisms is typically left to the system manager. Unfortunately, few individuals have the skills to properly manage the trade-offs involved, let alone the time to adapt these decisions to changing conditions. Instead, most systems are configured statically and with only a cursory understanding of how the configuration will impact overall performance or availability. While this issue can be problematic even for individual storage arrays, it becomes increasingly important as systems are distributed – and absolutely critical for the wide-area peer-to-peer storage infrastructures being explored.

This paper describes the motivation, architecture and implementation for a new peer-to-peer storage system, called *TotalRecall*, that automates the task of availability management. In particular, the *TotalRecall* system automatically measures and estimates the availability of its constituent host components, predicts their future availability based on past behavior, calculates the appropriate redundancy mechanisms and repair policies, and delivers user-specified availability while maximizing efficiency.

1 Introduction

Availability is a storage system property that is highly desired in principle, yet poorly understood in practice. How much availability is necessary, over what period of time and at what granularity? How likely are failures now and in the future and how much redundancy is needed to tolerate them? When should repair actions be initiated and how should they be implemented? These are all questions that govern the availability of a storage system, but they are rarely analyzed in depth or used to influence the dynamic behavior of a system.

Instead, system designers typically implement a static set of redundancy and repair mechanisms simply parameterized by resource consumption (e.g., number of replicas). Determining how to configure the mechanisms and what level of availability they will provide if employed is left for the user to discover. Moreover, if the underlying environment changes, it is again left to the user to reconfigure the system to compensate appropriately. While this approach may be acceptable when failures are consistently rare, such as for the individual drives in a disk array (and even here the management burden may be objectionable [23]), it quickly breaks down in large-scale distributed systems where hosts are transiently inaccessible and individual failures are common.

Peer-to-peer systems are particularly fragile in this respect as their constituent parts are in a continual state of flux. Over short time scales (1-3 days), individual hosts in such systems exhibit highly transient availability as their users join and leave the system at will – frequently following a rough diurnal pattern. In fact, the majority of hosts in existing peer-to-peer systems are inaccessible at any given time, although most are available over longer time scales [4, 19]. Over still longer periods, many of these hosts leave the system permanently, as most peer-to-peer systems experience high levels of churn in their overall membership. In such systems, we contend that availability management must be provided by the system itself, which can monitor the availability of the underlying host population and adaptively determine the appropriate resources and mechanisms required to provide a specified level of availability.

This paper describes the architecture, design and implementation of a new peer-to-peer storage system, called *TotalRecall*, that automatically manages availability in a dynamically changing environment. By adapting the degree of redundancy and frequency of repair to the distribution of failures, *TotalRecall* guarantees user-specified levels of availability while minimizing the overhead needed to provide these guarantees. We rely on three key approaches in providing these services:

- *Availability Prediction.* The system continuously monitors the current availability of its constituent hosts. This measured data is used to construct predictions, at multiple time-scales, about the future availability of individual hosts and groups of hosts.

- *Redundancy Management.* Short time-scale predictions are then used to derive precise redundancy requirements for tolerating transient disconnectivity. The system selects the most efficient redundancy mechanism based on workload behavior and system policy directives.

- *Dynamic Repair*. Long time-scale predictions coupled with information about current availability drive system repair actions. The repair policy is dynamically selected as a function of these predictions, target availability and system workload.

TotalRecall is implemented in C++ using a modified version of the DHash peer-to-peer object location service [8]. The system implements a variety of redundancy mechanisms (including replication and online coding), availability predictors and repair policies. However, more significantly, the system provides interfaces that allow new mechanisms and policies to describe their behavior in a unified manner – so the system can decide how and when to best use them.

The remainder of this paper describes the motivation, architecture and design of the *TotalRecall* system. The following section motivates the problem of availability management and describes key related work. In Section 3 we discuss our availability architecture – the mechanisms and policies used to ensure that user availability requirements are met. Sections 4 describes the design of the *TotalRecall* Storage System and its implementation. Section 5 describes the *TotalRecall* File System, a NFS file service implemented on the core storage system. In Section 6, we quantitatively evaluate the effectiveness of our system and compare it against existing approaches. Finally, Section 7 concludes.

2 Motivation and Related Work

The implicit guarantee provided by all storage systems is that data, once stored, may be recalled at some future point. Providing this guarantee has been the subject of countless research efforts over the last twenty years, and has produced a wide range of technologies ranging from RAID to robotic tape robots. However, while the efficiency of these techniques has improved over time and while the cost of storage itself has dropped dramatically, the complexity of managing this storage and its availability has continued to increase. In fact, a recent survey analysis of cluster-based services suggests that the operational cost of preparing for and recovering from failures easily dominates the capital expense of the individual hardware systems [17]. This disparity will only continue to increase as hardware costs are able to reflect advances in technology and manufacturing while management costs only change with increases in human productivity. To address this problem, the management burden required to ensure availability must be shifted from individual system administrators to the systems themselves. We are by no means the first to make this observation.

A major source of our inspiration is early work invested by HP into their AutoRAID storage array [23]. The AutoRAID system provided two implementations of storage redundancy – mirroring and RAID5 – and dynamically assigned data between them to optimize the performance of the current workload. While this system did not directly provide users with explicit control over availability it did significantly reduce the management burden associated with configuring these high-availability storage devices. A later HP project, AFRAID, did allow user-specified availability in a disk array environment, mapping availability requests into variable consistency management operations [20].

In the enterprise context, several researchers have recently proposed systems to automate storage management tasks. Keeton and Wilkes have described a system designed to automate data protection decisions that is similar in motivation to our own, but they focus on longer time scales since the expected failure distribution in the enterprise is far less extreme than in the peer-to-peer environment [10]. The WiND system is being designed to automate many storage management tasks in a cluster environment, but is largely focused on improving system performance [2]. Finally, the PASIS project is exploring system support to automatically make trade-offs between different redundancy mechanisms when building a distributed file system [24].

Not surprisingly, perhaps the closest work to our own arises from the peer-to-peer (P2P) systems community. Due to the administrative heterogeneity and poor host availability found in the P2P environment, almost all P2P systems provide some mechanism for ensuring data availability in the presence of failures. For example, the CFS system relies on a static replication factor coupled with an active repair policy, as does Microsoft's FARSITE system (although FARSITE calculates the replication factor as a function of total storage and and is more careful about replica placement) [1,8,9]. The Oceanstore system uses a combination of block-level erasure coding for long term durability and simple replication to tolerate transient failures [12,22]. Finally, a recent paper by Blake and Rodrigues argues that the cost of dynamic membership makes cooperative storage infeasible in the transiently available peer-to-peer environments [5]. This finding is correct under certain assumptions, but is not critical in the environments we have measured and the system we have developed.

What primarily distinguishes *TotalRecall* from previous work is that we allow the user to specify a specific availability target and then automatically determine the best mechanisms and policies to meet that request. In this way, *TotalRecall* makes availability a first-class storage property – one that can be managed directly and without a need to understand the complexities of the underlying system infrastructure.

3 Availability Architecture

There are three fundamental parameters that govern the availability of any system: the times at which components *fail* or become unavailable, the amount of redundancy employed to tolerate these outages and the time to detect and repair a failure.

The first of these is usually considered an independent parameter of the system, governed primarily by the environment and external forces not under programmatic control.[1] The remaining variables are dependent – they can be controlled, or at least strongly influenced, by the system itself. Therefore, providing a given level of availability requires predicting the likelihood of component failures and determining how much redundancy and what kind of repair policies will compensate appropriately.

The remainder of this section discusses the importance of these issues in turn, how they can be analyzed and how they influence system design choices. The following section then describes our concrete design and implementation of this overall architecture.

3.1 Availability Prediction

At the heart of any predictive approach to availability is the assumption that past behavior can be used to create a stochastic model of future outcomes. For example, "mean-time-to-failure" (MTTF) specifications for disk drives are derived from established failure data for similar components over a given lifetime. This kind of prediction can be quite accurate when applied to a large group of fail-stop components. Consequently, the future availability of single homogeneous disk arrays can be statically analyzed at configuration time.

However, in a distributed storage system – particularly one with heterogeneous resources and administration – this prediction can be considerably more complex. First, since the hosts composing such systems are rarely identical, the availability distribution cannot be analytically determined *a priori* – it must be measured empirically. Second, unlike disks, individual hosts in a distributed system are rarely fail-stop. Instead, hosts may be transiently unavailable due to network outages, planned maintenance or other local conditions. Consequently, such hosts may become unavailable and then return to service without any data loss. Finally, as such systems evolve, the number of hosts populating the system may grow or shrink – ultimately changing the availability distribution as well.

Nowhere is this issue more pronounced than in the public peer-to-peer environment. In such systems, the availability of an individual host is governed not only by failures, but more importantly by user decisions to disconnect from the network. Several recent studies of peer-to-peer activity have confirmed that individual hosts come and go at an incredible rate. In one such study of hosts in the Overnet system, we have observed that each host joined and left the system over 6 times per day on average [4]. In a similar study of Kazaa and Gnutella, Saroiu et al. found that the median session duration of a peer-to-peer system was only 60 minutes [19]. In addition to the transient availability found in these systems, public peer-to-peer populations exhibit a high rate of long-term churn as well. The previously-mentioned Overnet study found that approximately 20 percent of active hosts permanently departed from the system each day and roughly the same number of new hosts joined as well.

Consequently, in the peer-to-peer context, storage systems face two prediction requirements. First, the system must empirically measure the short-term availability distribution of its host population on an ongoing basis. We use this to model the probability of transient disconnections – those typically having no impact on the durability of data stored on disconnected hosts. From this distribution we estimate the *current* likelihood that a set of hosts will be available at any given time and subsequently determine the proper amount of redundancy needed. Our second prediction requirement focuses on non-transient failures that take stored data out of the system for indefinite periods. Since hosts are leaving the system continuously, redundancy is insufficient to ensure long-term storage availability. Instead the system must predict when hosts have "permanently" left the system (at least for long enough a period that they were no longer useful in the short term) and initiate a repair process.

3.2 Redundancy Management

In a peer-to-peer environment, each host may only be transiently available. When connected, the data stored on a host contributes to the overall degree of redundancy and increases the data's availability; when disconnected, both the degree of redundancy and data availability decreases. With sufficient redundancy across many hosts, at any moment enough hosts will be in the system to make a given data item available with high probability. However, it is not trivially clear how much redundancy is necessary for a given level of availability or what redundancy mechanism is most appropriate for a given context. We discuss both issues below.

There are a wide range of mechanisms available for producing redundant representations of data. However, each mechanism has unique trade-offs. For example, the simplest form of redundancy is pure replication. It has low run-time overhead (a copy) and permits efficient random access to sub-blocks of an object. However, replication can be highly inefficient in low-availability environments since many storage replicas are required to tolerate

[1] This is not always true, since processes that impact human error or opportunities for correlated failures can have an impact. However, we consider these issues outside the scope of this paper.

potential transient failures. At the other extreme, optimal erasure codes are extremely efficient. For a constant factor increase in storage cost, an erasure-coded object can be recovered at any time using a subset of its constituent blocks. However, the price for this efficiency is a quadratic coding time and a requirement that reads and writes require an operation on the entire object. By comparison, "non-optimal" erasure codes sacrifice some efficiency for significantly reduced on-line complexity for large files. Finally, it is easy to conceive of hybrid strategies as well. For example, a large log file written in an append-only fashion, might manage the head of the log using replication to provide good performance and eventually migrate old entries into an erasure coded representation for provide higher efficiency.

However, for all of these representations another question remains: how much redundancy is required to deliver a specified level of availability. More precisely: given an known distribution for short-term host availability and a target requirement for instantaneous data availability, how should these mechanisms be parameterized? Below we provide analytic approximations to these questions for pure replication and pure erasure coding. In both cases, our approach assumes that host failures are independent over short time scales. In previous work, we have provided a detailed explanation of our stochastic analysis and its assumptions [3], as well as experimental evidence to support our independence assumption [16]. Consequently, the *TotalRecall* system is not designed to survive catastrophic attacks or widespread network failures, but rather the availability dynamics resulting from localized outages, software crashes, disk failures and user dynamics.

Replication. Given a target level of availability A (where A represents the probability a file can be accessed at any time) and a mean host availability of μ_H, we can calculate the number of required replicas, c, directly.

$$A = 1 - (1 - \mu_H)^c \tag{1}$$

Solving for c,

$$c = \frac{log(1 - A)}{log(1 - \mu_H)} \tag{2}$$

Consequently, if mean host availability is 0.5, then it requires 10 complete copies of each file to guarantee a target availability of 0.999.

Some systems may choose to perform replication for individual blocks, rather than the whole file, as this allows large files to be split and balanced across hosts. However, this is rarely an efficient solution in a low-availability environment since every block (and hence at least one host holding each block) must be available for the file to be available. To wit, if a file is divided into b blocks, each of which has c copies, then the availability of that file is given by:

$$A = (1 - (1 - \mu_H)^c)^b \tag{3}$$

Consequently, a given level of availability will require geometrically more storage (as a function of b) in the block-level replication context.

Erasure coding. Given the number of blocks in a file b, and the stretch factor c specifying the erasure code's redundancy (and storage overhead) we can calculate the delivered availability as:

$$A = \sum_{j=b}^{cb} \binom{cb}{j} {\mu_H}^j (1 - \mu_H)^{(cb-j)} \tag{4}$$

If cb is moderately large, we can use the normal approximation to the binomial distribution to rewrite this equation and solve for c as:

$$c = \left(\frac{k\sqrt{\frac{\mu_H(1-\mu_H)}{b}} + \sqrt{\frac{k^2\mu_H(1-\mu_H)}{b} + 4\mu_H}}{2\mu_H} \right)^2 \tag{5}$$

More details on this equation's derivation can be found in [3]. For the same 0.999 level of availability used in the example above, an erasure-coded representation only requires a storage overhead of 2.49.

3.3 Dynamic Repair

However, the previous analyses only consider short-term availability – the probability that at a given instant there is sufficient redundancy to mask transient disconnections and failures. Over longer periods, hosts permanently leave the system and therefore the degree of redundancy afforded to an object will strictly decrease over time – ultimately jeopardizing the object's availability. In response, the system must "repair" this lost redundancy by continuously writing additional redundant data onto new hosts.

The two key parameters in repairing file data are the degree of redundancy used to tolerate availability transients and how quickly the system reacts to host departures. In general, the more redundancy used to store file data, the longer the system can delay before reacting to host departures.

Below we describe a spectrum of repair policies defined in terms of two extremes: eager and lazy. Eager repair uses a smaller degree of redundancy to maintain file availability guarantees by reacting to host departures immediately, but at the cost of additional communication overhead. In contrast, lazy repair uses additional redundancy, and therefore additional storage overhead, to delay repair and thereby reduce communication overhead.

3.3.1 Eager Repair

Many current research peer-to-peer storage systems maintain data redundancy pro-actively as hosts depart from the system. For example, the DHash layer of CFS replicates each block on five separate hosts [8]. When DHash detects that one of these hosts has left the system, it immediately repairs the diminished redundancy by creating a new replica on another host.

We call this approach to maintaining redundancy *eager repair* because the system immediately repairs the loss of redundant data when a host fails. Using this policy, data only becomes unavailable when hosts fail more quickly than they can be detected and repaired.

The primary advantage of eager repair is its simplicity. Every time a host departs, the system only needs to place redundant data on another host in reaction. Moreover, detecting host failure can be implemented in a completely distributed fashion since it isn't necessary to coordinate information about which hosts have failed. However, the eager policy makes no distinction between permanent departures that require repair and transient disconnections that do not. Consequently, in public peer-to-peer environments, many repair actions may be redundant and wasteful. In Section 6 we show via simulation that this overhead is very high for contemporary peer-to-peer host populations.

3.3.2 Lazy Repair

An alternative to eager repair is to defer immediate repair and use additional redundancy to mask and tolerate host departures for an extended period.[2] We call this approach *lazy repair* since the explicit goal is to delay repair work for as long as possible. The key advantage of lazy repair is that, by delaying action, it can eliminate the overhead of redundant repairs and only introduce new redundant blocks when availability is threatened.

However, lazy repair also has disadvantages. In particular, it must explicitly track the availability of individual hosts and what data they carry. This is necessary to determine when an object's availability is threatened and a repair should be initiated. Consequently, the system must maintain *explicit* metadata about which hosts hold what data. By contrast, eager implementations can make use of the implicit mappings available through mechanisms like consistent hashing [8]. For small objects, this can significantly increase the overhead of repair actions.

For lazy repair, the system must incorporate additional redundancy for files so that it can tolerate host departures over an extended period. Hence while the analysis in the previous section gives us the *short-term* redundancy factor used to tolerate transient failures, each file needs to use a larger *long-term* redundancy factor to accommodate host failures without having to perform frequent file repairs.

[2]This is similar, in spirit, to Oceanstore's *refresh* actions which are meant to ensure data durability in the face of disk failures [12].

As mentioned in Section 3.1, the system requires an availability predictor that will estimate when a file needs to be repaired. A simple predictor for lazy repair periodically checks the total amount of available redundancy for a given file. If this value falls below the short-term redundancy factor for the file, then the system triggers a repair. Thus we also refer to the short-term redundancy factor as the *repair threshold* for the file.

Section 6 compares the repair bandwidth required by each policy using an empirical trace of peer-to-peer host availability patterns.

3.4 System Policies

The combination of these mechanisms – prediction, redundancy and repair – must ultimately be combined into a system-wide strategy for guaranteeing file availability. Minimally, a system administrator must specify a file availability target over a particular lifetime. From these parameters, coupled with an initial estimate of host availability, an appropriate level of redundancy can be computed. In addition to repair actions triggered by the disappearance of individual hosts, the system may occasionally need to trigger new repair actions to compensate for changes in the overall availability of the entire population. For example, a worm outbreak may reduce the average host availability system-wide or the expansion of broadband access may increase the average uptime of connected hosts.

However, there is significant room for more advanced policies. For example, there is a clear trade-off between random access performance and storage efficiency in the choice of redundancy mechanism. A system policy can make this trade-off dynamically in response to changing workloads. For instance, a file might use an erasure coded base representation, but then replicate frequently accessed sub-blocks independently. As well, system policies could easily specify different availability requirements for different portions of the file system and even calculate availability as a function of file dependencies (e.g., a user may wish to request a given level of availability for the combination of the mail program and the mail spool it uses).

4 *TotalRecall* Storage System

This section describes the design and implementation of the *TotalRecall* Storage System. The *TotalRecall* Storage System implements the availability architecture described in Section 3 in a cooperative host environment. It provides a simple read/write storage interface

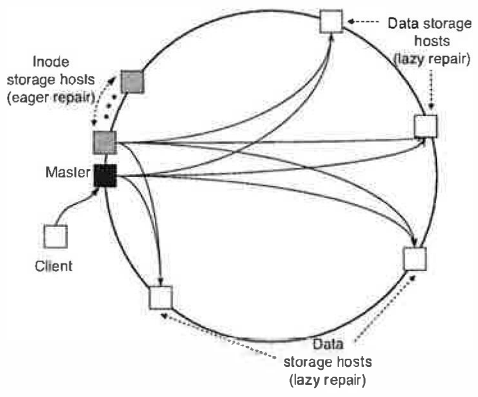

Data storage hosts (lazy repair)
Inode storage hosts (eager repair)
Master
Client
Data storage hosts (lazy repair)

Figure 1: *TotalRecall* system architecture.

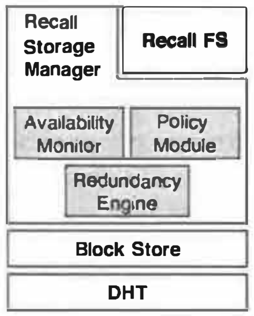


Figure 2: *TotalRecall* host architecture.

for maintaining data objects with specified target availability guarantees.

4.1 System Overview

Hosts in *TotalRecall* contribute disk resources to the system and cooperatively make stored data persistent and available. Figure 1 shows an overview of *TotalRecall* with participating hosts organized in a ring ID space. *TotalRecall* stores and maintains *data objects*, conveniently referred to as *files*. Files are identified using unique IDs. The system uses these IDs to associate a file with its *master* host, the host responsible for maintaining the persistence, availability, and consistency of the file. *Storage* hosts persistently store file data and metadata according to the repair policy the master uses to maintain file availability. *Client* hosts request operations on a file. Clients can send requests to any host in the system, which routes all requests on a file to its master. As a cooperative system, every *TotalRecall* host is a master for some files and storage host for others; hosts can also be clients, although clients do not need to be *TotalRecall* hosts.

A *TotalRecall* server runs on every host in the system. As shown in Figure 2, the *TotalRecall* host architecture has three layers. The *TotalRecall* Storage Manager handles file requests from clients and maintains file availability for those files for which it is the master. It uses the Block Store layer to read and write data blocks on storage hosts. The Block Store in turn uses an underlying distributed hash table (DHT) to maintain the ID space and provide scalable lookup and request routing.

4.2 Storage Manager

The *TotalRecall* Storage Manager (TRSM) implements the availability architecture described in Section 3. It has three components, the policy module, the availability monitor, and the redundancy engine (see Figure 2).

The TRSM invokes the policy module when clients create new files or substantially change file characteristics such as size. The policy module determines the most efficient strategy for maintaining stored data with a target availability guarantee. The strategy is a combination of redundancy mechanism, repair policy, and number of blocks used to store coded data. It chooses the redundancy mechanism (e.g., erasure coding vs. whole-file replication) based on workload characteristics such as file size and the rate, ratio, and access patterns of read and write requests to file data (Section 3.4). The repair policy determines how the TRSM maintains data availability over long-term time scales to minimize repair bandwidth for a target level of availability (Section 3.3). Although redundancy and repair are orthogonal, for typical workloads *TotalRecall* uses replication and eager repair for small files and erasure coding and lazy repair for large files (Section 6.3.1). Finally, with lazy repair the policy module also determines the number of blocks to use with erasure coding to balance file availability and communication overhead; more blocks increases availability but requires the TRSM to contact more storage hosts to reconstruct the file [3].

The TRSM dynamically adapts its mechanisms and policies for efficiently maintaining data according to the availability of hosts in the system. To do this, the availability monitor (AM) tracks host availability, maintains host availability metrics that are used by other components, and notifies the redundancy engine when the system reaches availability thresholds that trigger repair. The AM tracks the availability of the storage hosts storing the metadata and data for those files for which it is the master. Based upon individual host availability, the AM maintains two metrics: *short-term host availability* and *long-term decay rate* (Section 3.1). Short-term host availability measures the minimum average of all tracked hosts that were available at any given time in the past 24 hours (e.g., 50% of hosts were available at 4am). It is a conservative prediction of the number of hosts available over the course of a day. Long-term decay rate measures the rate at which hosts leave the system over days and weeks, and is used to predict the frequency of repair. Finally, the TRSM registers to receive events from the AM whenever the availability of a set of storage hosts drops below a specified threshold to trigger repairs.

Whereas the policy module decides what kind of redundancy mechanism to use based upon high-level workload characteristics, the redundancy engine (RE) implements the redundancy mechanisms and determines how much short-term and long-term redundancy to use for a file based upon current system conditions. The TRSM invokes the redundancy engine when writing and repairing files. The RE currently supports simple replication and erasure coding. For replication, the RE uses Equation 2 in Section 3.2 to determine the number of replicas r to create when storing the file. It uses the target availability A associated with the file and the short-term host availability from the AM as inputs to the equation. For erasure coding, the RE uses Equation 5 to determine the short-term redundancy r (also called repair threshold) for encoding the file. It uses the target availability A associated with the file, the short-term host availability from the AM, and the number of blocks b determined by the policy module as inputs to the equation.

4.3 Storage Layout

For every file, the *TotalRecall* Storage Manager uses *inodes* as metadata to locate file data and maintain file attributes such as target availability, size, version, etc. It stores inodes for all files using replication and eager repair. The master stores inodes itself and a set of replicas on its successors, much like DHash blocks [8], and the redundancy engine determines the number of replicas (Section 4.2). Figure 1 shows an example of storing an inode for a lazily repaired file. The master updates inodes in place, and it serializes all operations on files by serializing operations on their inodes (e.g., a write does not succeed until all inode replicas are updated).

The TRSM stores data differently depending upon the repair policy used to maintain file availability. For files using eager repair, the TRSM on the master creates a unique file data ID and uses the DHT to lookup the storage host responsible for this ID. It stores file data on this storage host and its successors in a manner similar to inodes. The inode for eagerly repaired files stores the file data ID as a pointer to the file data.

For files using lazy repair, the TRSM stores file data on a randomly selected set of storage hosts (Section 3.3.2). Figure 1 also shows how the master stores file data for lazily repaired files. It stores the IDs of the storage hosts in the file's inode to explicitly maintain pointers to all of the storage hosts with the file's data. It also uses the redundancy engine to determine the number of storage hosts to use, placing one block (erasure coding) or replica (replication) per storage host.

File data is immutable. When a client stores a new version of a file that is lazy repaired, for example, the TRSM randomly chooses a new set of storage hosts to store the data and updates the file's inode with pointers to these hosts. The TRSM uses the version number stored in the inode to differentiate file data across updates. A garbage collection process periodically reclaims old file data, and a storage host can always determine whether its file data is the latest version by looking up the inode at the master (e.g., when it joins the system again after being down).

4.4 Storage API

The *TotalRecall* Storage Manager implements the storage API. The API supports operations for creating, opening, reading, writing, and repairing files, and similar operations for inodes. All request operations on a file are routed to and handled by the file's master. Lacking the space to detail all operations, we highlight the semantics of a few of them.

Clients use `tr_create` to create new files, specifying a target availability A for the file upon creation. It is essentially a metadata operation that instantiates a new inode to represent the file, and no data is stored until a write operation happens. `tr_read` returns file data by reading data from storage hosts, decoding erasure-coded data when appropriate. `tr_write` stores new file data or logically updates existing file data. It first sends the data to storage hosts and then updates the inode and inode replicas (see Section 4.5). For lazily repaired files, encoding and distributing blocks for large files can take a considerable amount of time. To make writes more responsive, the master uses a background process that performs the encoding and block placement offline. The master initially eagerly repairs the blocks using simple replication, and then erasure codes and flushes these blocks out to the storage hosts.

The TRSM also implements the `tr_repair` operation for repairing file data, although its execution is usually only triggered internally by the availability manager. For eager repair, `tr_repair` repairs data redundancy immediately when a host storing data departs. For lazy repair, it only repairs data when the number of hosts storing file data puts the file data at risk of violating the file's target availability. Since this occurs when much of the file's data is on hosts that are not available, `tr_repair` essentially has the semantics of a file read followed by a write onto a new set of hosts.

4.5 Consistency

Since the system maintains replicas of inodes and inodes are updated in place, the master must ensure that inode updates are consistent. In doing so, the system currently assumes no partitions and that the underlying DHT provides consistent routing — lookups from different hosts for the same ID will return the same result.

When writing, the master ensures that all data writes complete before it updating the inode. The master writes all redundant data to the storage hosts, but does not start updating the inode until all the storage hosts have acknowledged their writes. If a storage host does fail during the write, the master will retry the write on another storage host. Until all data writes complete, all reads will see the older inode and, hence, the older version of the file. As the master makes replicas of the inode on its successors, it only responds that the write has completed after all successors have acknowledged their writes. Each inode stores a version number assigned by the master ordered by write requests to ensure consistent updates to inode replicas. Once a successor stores an inode replica, eager repair of the inode ensures that the replica remains available. If the master fails as it updates inodes, the new master will synchronize inode versions with its successors. If the master fails before acknowledging the write, the host requesting the write will time out and retry the write to the file. A new master will assume responsibility for the file, receive the write retry request, and perform the write request. As a result, once a write completes, i.e., the master has acknowledged the write to the requester, all subsequent reads see the newest version of the file.

4.6 Implementation

We have implemented a prototype of the *TotalRecall* storage system on Linux in C++. The system consists of over 5,700 semi-colon lines of new code. We have also reused existing work in building our system. We use the SFS toolkit [14] for event-driven programming and MIT's Chord implementation as the underlying DHT [21]. Files stored using eager repair use a modified version of the DHash block store [8].

The prototype implements all components of the *TotalRecall* Storage Manager, although some advanced behavior remains future work. The prototype policy module currently chooses the redundancy mechanism and repair policy solely based on file size: files less than 32 KB use replication and eager repair, and larger files use erasure coding and lazy repair. For lazy repair, files are fragmented into a minimum of 32 blocks with a maximum block size of 64 KB. To erasure code lazily repaired files, the redundancy engine implements Maymounkov's online codes [13], a sub-optimal linear-time erasure-coding algorithm. The redundancy engine also uses a default constant long-term redundancy factor of 4 to maintain lazy file availability during the repair period.

The availability monitor tracks host availability by periodically probing storage hosts with an interval of 60 seconds. This approach has been sufficient for our experiments on PlanetLab, but would require a more scalable approach (such as random subsets [11]) for tracking and disseminating availability information in large-scale deployments. The TRSM uses the probes to storage hosts for a file to measure and predict that file's availability. Based upon storage host availability, the TRSM calculates the amount of *available redundancy* for the file. The available redundancy for the file is the ratio of the total number of available data blocks (or replicas) to the total number of data blocks (replicas) needed to read the file in its entirety. When this value drops below the repair threshold, the AM triggers a callback to the TRSM, prompting it to start repairing the file. The prototype by default uses a repair threshold of 2. With a long-term redundancy factor of 4 for lazy repair, for example, when half of the original storage hosts are unavailable the AM triggers a repair.

In building our prototype, we have focused primarily on the issues key to automated availability management, and have not made any significant effort to tune the system's runtime performance. Addressing run-time overheads, as well as implementing more advanced performance and availability tradeoffs in the policy module, remains ongoing work.

5 *TotalRecall* File System

The *TotalRecall* Storage System provides a core storage service on which more sophisticated storage services can be built, such as backup systems, file-sharing services, and file systems. We have designed one such service, the *TotalRecall* File System (TRFS), an NFSv3-compatible file system [7]. To provide this service, the *TotalRecall* File System extends the Storage Manager with the TRFS Manager (see Figure 2). The TRFS Manager extends the storage system with file system functionality, implementing a hierarchical name space, directories, etc. It extends the *TotalRecall* Storage Manager with an interface that roughly parallels the NFS interface, translating file system operations (e.g., mkdir) into lower-level TRSM operations.

Clients use a TRFS loopback server to mount and access TRFS file systems. The loopback server runs on the client as a user-level file server process that supports the NFSv3 interface [14]. It receives redirected NFS operations from the operating system and translates them into RPC requests to *TotalRecall*.

We have implemented TRFS as part of the *TotalRecall* prototype, adding 2,000 lines of code to implement the TRFS Manager and loopback server. It currently supports all NFSv3 operations except hard links.

6 Experimental Evaluation

In this section, we evaluate *TotalRecall* using both trace-driven simulation and empirical measurements of our prototype implementation. We use simulation to study the effectiveness of our availability predictions, the be-

Hosts	5500
Files	5500
No. of Blocks Before Encoding	32
File Sharing File Sizes	4 MB (50%), 10 MB (30%), 750 MB (20%)
File System File Sizes	256 B (10%), 2 KB (30%), 4 KB (10%), 16 KB (20%), 128 KB (20%), 1 MB (10%)

Table 1: File workloads used to parameterize simulation.

havior of the system as it maintains file availability, the tradeoffs among different repair policies, and *TotalRecall*'s use of bandwidth resources to maintain file availability. And we evaluate the prototype implementation of the *TotalRecall* File System on a 32-node deployment on PlanetLab and report both per-operation microbenchmarks and results from the modified Andrew benchmark.

6.1 Simulation Methodology

Our simulator, derived from the well-known Chord simulation software [21], models a simple version of the *TotalRecall* Storage System. In particular, it models the availability of files across time as well as the bandwidth and storage used to provide data and metadata redundancy. The simulator is designed to reveal the demands imposed by our system architecture and not for precise prediction in a particular environment. Consequently, we use a simple model for host storage (infinite capacity) and the network (fixed latency, infinite bandwidth, no congestion).

To drive the simulator we consider two different file workloads and host availability traces. The two file workloads, parameterized in Table 1, consist of a *File Sharing* workload biased towards large files [18] and a more traditional *File System* workload with smaller files [6]. Similarly, we use two corresponding host availability traces. The File Sharing trace is a scaled down version of a trace of host availability in the Overnet file sharing system [4], while the File System availability trace is synthetically generated using the host availability distribution in [6]. The two availability traces are both one week long and differ primarily in their dynamics: the File Sharing trace has an average host uptime of 28.6 hours, compared to 109 hours in the File System trace.

The simulations in this section execute as follows. Hosts join and leave the system, as dictated by the availability trace, until the system reaches steady-state (roughly the same number of joins as leaves). Then files are inserted into the system according to the file workload. Subsequent joins and leaves will cause the system to trigger repair actions when required. The system repairs inodes eagerly, and data eagerly or lazily depending on policy (Section 4.6). From the simulation we can then determine the delivered file availability and bandwidth usage: the two primary metrics we evaluate.

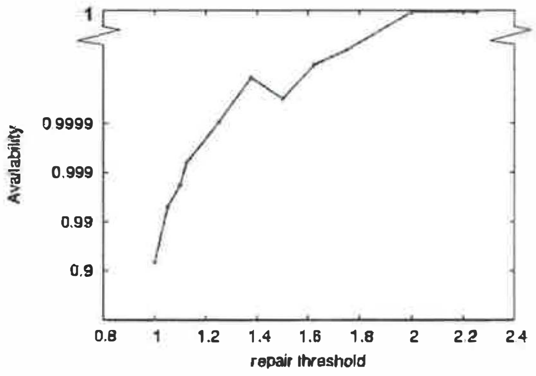


Figure 3: Empirical file availability calculated for the File Sharing host availability distribution.

6.2 Delivered Availability

It is critical that *TotalRecall* is able to deliver the level of availability requested. To verify, we specify a target availability of 0.99 and from this compute the required repair threshold. Using Equation 4 with an average host availability of 0.65, we compute that an erasure coded file with lazy repair will require a repair threshold of at least 2 to meet the availability target.

To see how well this prediction holds, we simulate a series of periodic reads to all 5500 files in the File Sharing workload. Using the associated host availability trace to drive host failures, we then calculate the average file availability as the ratio of completed reads to overall requests. Figure 3 shows how this ratio varies with changes in the repair threshold (this assumes a constant long-term redundancy factor of 4). From the graph, we see that files with a repair threshold of 2 easily surpass our 0.99 availability target. For this trace a lower repair threshold could also provide the same level of availability, although doing so would require more frequent file repairs.

To provide better intuition for this dynamic, Figure 4 shows the repair behavior of *TotalRecall* over time at a granularity of 60 minutes. We use the File Sharing workload parameters in Table 1 to parameterize the system and the File Sharing host availability trace to model host churn. The three curves on the graph show the number of available hosts, the bandwidth consumed by the system, and the average *normalized available file redundancy* across all files in each time interval. Available file redundancy measures the amount of redundant data that the system has available to it to reconstruct the file. For each file, we normalize it with respect to the long-term redundancy factor used for the file, so that we can compute an average over all files.

Looking at the curves over time, we see how *TotalRecall* uses lazy repair to maintain a stable degree of data

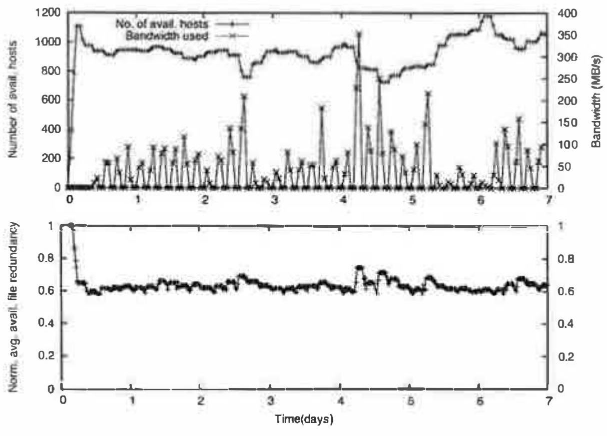


Figure 4: System behavior on the File Sharing workload.

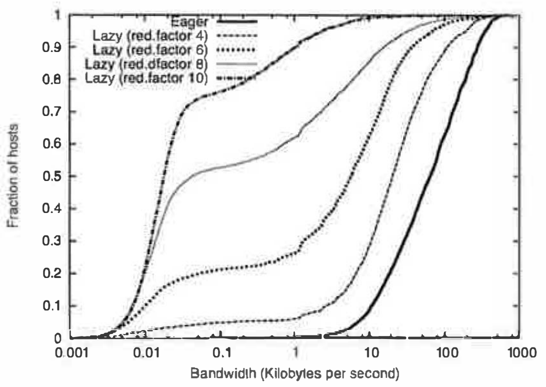


Figure 5: CDF of the bandwidth usage of hosts in *TotalRecall* for different repair policies.

redundancy as host availability varies substantially over time. Note that though the total number of hosts available at any time is roughly between 800 and 1000, new hosts are constantly joining the system while old hosts leave, causing substantial amounts of host churn. We make three observations of the system behavior.

First, we see that system bandwidth varies with host availability. As hosts leave the system, *TotalRecall* eagerly repairs inodes and lazily repairs data blocks for those files whose predicted future availability drops below the lazy repair threshold. Consequently, relative system bandwidth increases as the number of hosts decreases. As hosts join the system, *TotalRecall* eagerly repairs inodes but does not need to repair data blocks. Consequently, relative system bandwidth decreases as the number of hosts increases.

Second, the normalized average degree of available redundancy reaches and maintains a stable value over time, even with substantial host churn. This behavior is due to the design of the lazy repair mechanism. Files stored using a lazy repair policy experience cyclic behavior over time. When the system first stores a file using lazy repair, it places all of the redundant data blocks on available hosts. At this time, the file has maximum available redundancy (since we create all files at time 0, all files have maximum available redundancy at time 0 in Figure 4). As hosts leave the system, file blocks become unavailable. As hosts join the system again, file blocks become available again. As a result, available file redundancy fluctuates over time. But the long-term trend is for blocks to become unavailable as hosts depart the system for extended periods of time (possibly permanently). Eventually, based upon *TotalRecall*'s prediction of future host availability and current available file redundancy (Section 3.1), enough blocks will become inaccessible that the system will trigger a lazy repair to ensure continued file availability. Lazy repair will replace missing redundant blocks and raise available file redundancy back to its maximum, and the cycle continues.

Third, the overall average system repair bandwidth for the entire time duration is 35.6 MB/s. Dividing by the number of files, the average repair bandwidth per file is 6.5 KB/s. While this is not insignificant, we believe that, given the large file sizes in the File Sharing workload (20% 750MB), this figure is reasonably small. Also, note that using larger long-term redundancy factors has the effect of reducing the bandwidth usage of the system (Figure 5). Breaking down bandwidth overhead by use, overall 0.6% of the bandwidth is used for eager repair of inodes and 99.4% is used for lazy repair of data blocks

6.3 Repair Overhead

A key design principle of *TotalRecall* is to adapt the use of its repair policies to the state of the system. These policies have various tradeoffs among storage overhead, bandwidth overhead, and performance, and interact with the distributions of host availability and file sizes as well. Finally, *TotalRecall* efficiency hinges on accurate prediction of future failures. We investigate these issues in turn.

6.3.1 Repair Policy

To illustrate the tradeoff between storage and bandwidth, we simulate the maintenance of the File Sharing workload on *TotalRecall* and measure the bandwidth required to maintain file availability using the File Sharing host availability trace to model host churn. (Note that we do not include the bandwidth required to write the files for the first time.) We measure the average bandwidth required by each node to maintain its inode and data blocks across the entire trace for five different repair policies: eager repair, and lazy repair with erasure coded data using four different long-term redundancy factors.

Figure 5 shows the cumulative distribution function of the average bandwidth consumed by the hosts in the system over the trace for the different repair policies. From the graph, we see that eager repair requires the most

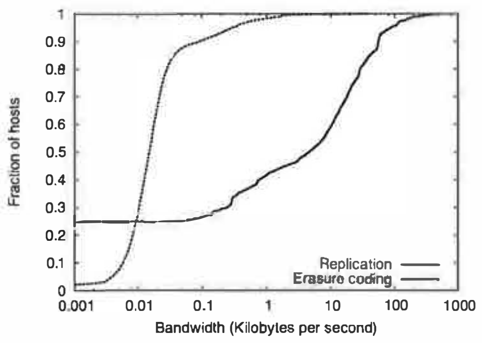


Figure 6: CDF of the bandwidth usage of hosts comparing replication and erasure coding for lazy repair.

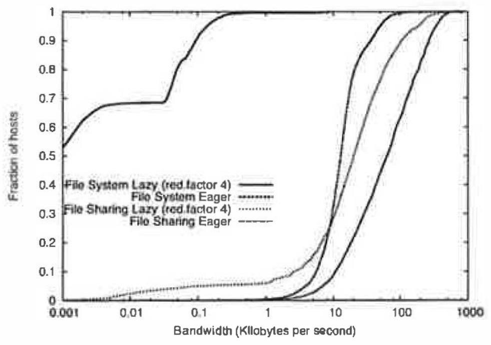


Figure 7: CDF of the bandwidth usage of hosts comparing eager and lazy repair on different workloads.

maintenance bandwidth across all hosts, lazy repair with a long-term redundancy factor of 4 requires the second-most bandwidth, and larger long-term redundancy factors require progressively less bandwidth. These results illustrate the fundamental tradeoffs between redundancy and repair bandwidth. Eager repair, which uses convenient but minimal redundancy, cannot delay repair operations and requires the most bandwidth. Lazy repair, which uses more sophisticated redundancy to delay repair, requires less bandwidth, especially with significant host churn as in the File Sharing scenario. Lazy repair with lower long-term redundancy factors require less storage, but more frequent repair. Higher long-term redundancy factors delay repair, but require more storage.

The shapes of the curves in Figure 5 show how the bandwidth requirements vary across all of the hosts in the system for the different repair policies. Eager repair essentially has a uniform distribution of bandwidth per node across all hosts. This is mainly due to the fact that hosts are assigned random IDs. Consequently, hosts leave and join the system at random points in the DHT, and the load of making replicas of inodes and file blocks is well distributed among all the hosts in the system.

In contrast, lazy repair essentially has two categories of hosts. The first is all the hosts that store some file data blocks. The second are hosts that join and leave the system before any file repairs are triggered, do not receive any file data, and participate only in the eager repair of inodes. As a result, the bandwidth usage of these hosts is smaller than those that store data. For larger long-term redundancy factors such as 8 and 10, file repairs are not that frequent, and hence there are a significant number of hosts that fall into the second category. Curves for these long-term redundancy factors in Figure 5 have a sharp rise around 30 bytes per second, demonstrating the presence of an increasing number of such hosts with increasing long-term redundancy factor.

So far we have concentrated on evaluating lazy repair with erasure coding. We now study how lazy repair with coding compares with lazy repair with replication. The question that this experiment seeks to answer is that for the same level of file availability and storage, how does the bandwidth usage of lazy repair with coding compare to the bandwidth usage of lazy repair with replication.

To maintain a file availability of 0.99, Equations 3 and 4 estimate that lazy repair with erasure coding has a repair threshold of 2 and lazy repair with replication requires 5 replicas. In other words, the system needs to repair files with erasure coding when the redundancy (degree of coded data) falls below 2, and the system would have to perform repairs with replication when the available redundancy (number of replicas) falls below 5. Lazy repair with replication therefore potentially uses more bandwidth than lazy repair with erasure coding.

To quantify how much more bandwidth replication uses, we repeat the bandwidth measurement simulation experiment but assign a long-term redundancy factor of 10 to each file. For lazy repair with coding, the system performs a file repair when the file redundancy falls to 2 and, for lazy repair with replication, the system repairs the file when the redundancy falls to 5. Figure 6 shows the CDF of bandwidth required per host for these two cases. From the graph, we see that the system bandwidth requirements to perform lazy repair with replication are far higher than that required for lazy repair with erasure codes. The average bandwidth per host for lazy repair with erasure coding is 655 Bps, while lazy repair with replication is 75 KBps. Our conclusion from these experiments is that for large file size distributions, and for highly dynamic and highly unavailable storage components, lazy repair with erasure coding is the more efficient availability maintenance technique.

6.3.2 Host Availability

To study the affect of different host availability distributions on bandwidth usage, we compared the bandwidth consumed for each host for the File Sharing host availability trace with that of the File System trace.

Figure 7 shows that the File System availability trace requires less bandwidth and that lazy repair works partic-

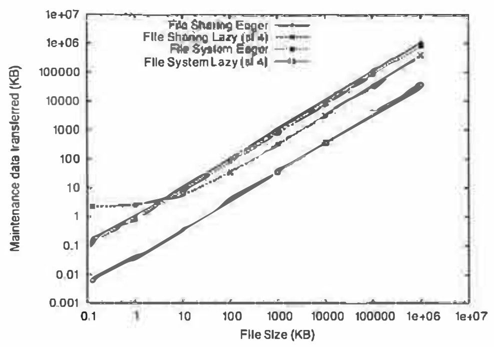


Figure 8: Per-file bandwidth required for repair.

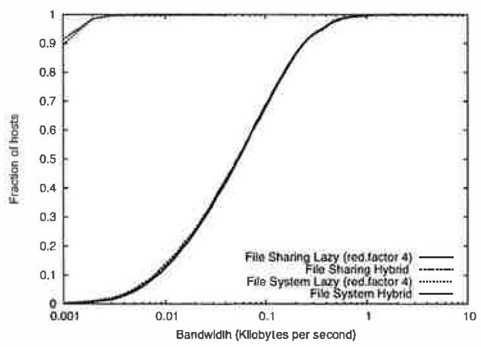


Figure 9: CDF of the bandwidth usage per host for lazy repair and the hybrid policy.

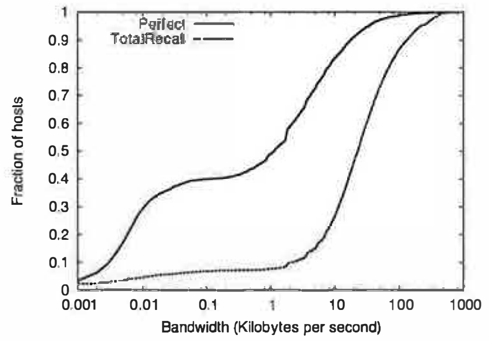


Figure 10: CDF of the bandwidth usage per host comparing *TotalRecall* with an optimal system.

ularly well. Since the availability of hosts is higher for this trace, the host churn is lower. Moreover, the source of this trace was collected from a workplace, and as a result we see a more cyclic pattern of availability for some hosts. These hosts contribute to eager repair bandwidth usage since they increase the churn in the system. However, since they cyclically re-appear in the system, they do not trigger lazy repair.

6.3.3 File Size

The repair policy for a file depends on file size (Section 4.2). To illustrate the tradeoff between eager and lazy repair on file size, we measure the bandwidth usage per file in the system with both eager and lazy repair, for both File Sharing and File System host availability traces, for various file sizes.

Figure 8 shows the average system bandwidth for maintaining each file for the entire trace for a range of file sizes. For each host availability trace, the graph shows two curves, one where the system maintains files using eager repair and the other where the system uses lazy repair. From the graph, we see that, for the File Sharing trace, eager repair requires less bandwidth to maintain small files less than approximately 4 KB in size, but that lazy repair requires less bandwidth for all larger files. This crossover between eager and lazy is due to the larger inodes required for lazy repair. For the File System trace however, we do not see a crossover point. Since the trace has less churn, fewer repairs are required and less bandwidth is consumed for eager repair. Eager repair is better for smaller file sizes and higher host availability.

To see the effect of using a hybrid repair policy, i.e., using eager repair for files smaller than 4 KB and lazy for all others, we simulated the File System workload on *TotalRecall* using the File Sharing and the File System host traces. Figure 9 shows the CDF of average bandwidth usage per host for pure lazy repair and the hybrid policy, and for both host availability traces. There is very little difference in the bandwidth usage between the two curves for the same host trace. From this we conclude that, for small files, *TotalRecall* should use eager repair. While bandwidth usage is comparable to that for lazy repair, the performance will be better since the system avoids the computational overhead of encoding/decoding these files and also avoids the communication overhead of distributing them over many storage hosts.

6.3.4 Prediction

Though we have established that lazy repair with erasure coding is the most efficient availability maintenance technique in our system, we would like to see how close *TotalRecall* comes to optimal bandwidth usage in the system. The question we address is, if there existed an Oracle that would repair a file *just* before it becomes unavailable, how would the system's bandwidth usage characteristics compare to those of *TotalRecall*?

To answer this question, we compare *TotalRecall*'s bandwidth consumption using lazy repair and erasure coding to that of an optimal system that also uses lazy repair and erasure coding. The optimal system minimizes bandwidth by performing repairs just before the files become unavailable. Note that a file becomes unavailable when its redundancy drops below 1 (less data available than originally in the file). To model the optimal system, we modified the simulator so that whenever the availability monitor detected that a file's redundancy dropped below 1, it would initiate a repair. In contrast, *TotalRecall*

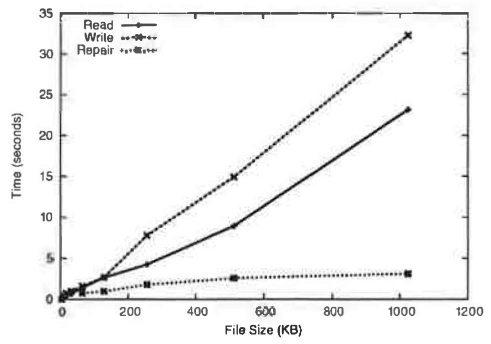


Figure 11: File system performance on PlanetLab.

initiates file repair whenever the file's redundancy drops just below the repair threshold of 2. Both *TotalRecall* and the optimal simulator use a long-term redundancy factor of 4 for this experiment.

Figure 10 shows the results of this experiment. The bandwidth usage in *TotalRecall* is almost an order of magnitude more than the optimal system. While the average bandwidth usage in *TotalRecall* is 49 KBps, the optimal system is 7 KBps. The difference is due to two reasons. First, it is very difficult to predict host behavior accurately given the strong dynamics in the system. Second, *TotalRecall*'s main goal is to guarantee availability of files, and in doing so, it tends to make conservative estimates of when file repairs are required. We believe this to be a suitable design decision given the system's goals. However, this experiment does show that there is room for the system to improve its bandwidth usage by using more sophisticated techniques to predict host failures and file availability.

6.4 Prototype Evaluation

The simulation experiments focused on the file availability and bandwidth overhead of providing available files in the *TotalRecall* Storage System. Next, we evaluate the performance of the prototype *TotalRecall* File System implementation. To perform our measurements, we ran TRFS on a set of 32 PlanetLab hosts distributed across the U.S. We used a local machine as a client host mounting the *TotalRecall* File System via the TRFS loopback server. In all experiments below, the system uses an eager repair threshold of 32 KB, i.e., the system replicates and eager-repairs all files smaller than 32 KB, and it uses erasure coding and lazy repair for all files of size greater than 32 KB. For lazy repair, the system uses a long-term redundancy factor of 4.

6.4.1 Microbenchmarks

We first evaluate the *TotalRecall* File System by measuring the performance of file system operations invoked via the NFS interface. Figure 11 shows the sequential file read, write, and repair performance for lazily repaired files in TRFS running on the 32 PlanetLab hosts for a range of file sizes. First, we measured the performance of using the NFS interface to write a file of a specified size using a simple C program. We then measured the performance of the same program reading the file again, making sure that no cached data was read. Finally, we forced the master node responsible for the file to perform a lazy repair. Lazy repair roughly corresponds to a combined read and write: the master node reads a sufficient number of file blocks to reconstruct the file, and then writes out a new encoded representation of the file to a new set of randomly chosen nodes.

From the graph, we see that write performance is the worst. Writes perform the most work: writing a file includes creating and storing inodes, encoding the file data, and writing all encoded blocks to available hosts. Read performance is better because the master node need only read a sufficient number of blocks to reconstruct the file. Since this number is smaller than the total number of encoded blocks stored during a write read operations require fewer RPC operations to reconstruct the file.

Finally, lazy repair performs the best of all. Although this might seem counterintuitive since the lazy repair operation requires more work than read or write, lazy repair operates within the *TotalRecall* Storage System. As a result, it is able to operate in parallel on much larger data aggregates than the read and write operations. NFS serializes read operations, for example, in 4KB block requests to the master. However, when the master reads blocks to perform a lazy repair, it issues 64KB block requests in parallel to the storage hosts.

Phase	Duration (s)
mkdir	12
create/write	60
stat	64
read	83
compile	163
Total	392

Table 2: Wide-area performance of the modified Andrew benchmark on 32 PlanetLab nodes.

6.4.2 Modi□ed Andrew Benchmarks

We also ran the modified Andrew benchmarks on TRFS on 32 hosts on PlanetLab. We chose hosts that were widely distributed across the U.S. Table 2 shows the results of running these benchmarks on *TotalRecall*. We see that the read phase of the benchmark takes longer than the write phase. Since the benchmark primarily consists of small files that are eagerly-repaired and replicated in our system, the writes take less time than if they were erasure coded. The compile phase, however, takes a fair amount of time since the final executable is large and it is erasure-coded and lazy-repaired. The total time of execution of the benchmarks was 392 seconds. As one point of rough comparison, we note that in [15] the authors evaluated the Ivy peer-to-peer file system on 4 hosts across the Internet using the same benchmark with a total execution time of 376 seconds.

The absolute performance of the *TotalRecall* File System is not remarkable, and not surprising since we have not focused on performance. In part this is due to the wide variance in the underlying network performance of the PlanetLab hosts used in our experiments (e.g., 25% of the nodes have RPC latency over 100 ms) and time spent in software layers underneath *TotalRecall* (e.g., 87% of the time writing 4 KB files in Figure 11 is spent in Chord lookups and block transfers from storage hosts). Given that our implementation is an unoptimized prototype, we are also exploring optimizations to improve performance, such as aggregating and prefetching data between clients and the master to improve NFS performance.

7 Conclusions

In this paper, we have argued that storage availability management is a complex task poorly suited to human administrators. This is particularly true in the large-scale dynamic systems found in peer-to-peer networks. In these environments, no single assignment of storage to hosts can provide a predictable level of availability over time and naive adaptive approaches, such as eager replication, can be grossly inefficient.

Instead, we argue that availability should become a first class system property – one specified by the user and guaranteed by the underlying storage system in the most efficient manner possible. We have proposed an architecture in which the storage system predicts the availability of its components over time, determines the appropriate level of redundancy to tolerate transient outages, and automatically initiates repair actions to meet the user's requirements. Moreover, we have described how key system parameters, such as the appropriate level of redundancy, can be closely approximated from underlying measurements and requirements. Finally, we described the design and implementation of a prototype of this architecture. Our prototype peer-to-peer storage system, called *TotalRecall*, automatically adapts to changes in the underlying host population, while effectively managing file availability and efficiently using resources such as bandwidth and storage.

Acknowledgments

We would like to thank the reviewers for their valuable comments, Barbara Liskov for being our shepherd, and Marvin McNett for the system support during the development of *TotalRecall*. Support for this work was provided in part by DARPA FTN Contract N66001-01-1-8933 and AFOSR MURI Contract F49620-02-1-0233.

References

[1] A. Adya et al. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of OSDI*, 2002.

[2] A. C. Arpaci-Dusseau et al. Manageable storage via adaptation in WiND. In *IEEE CCGrid*, 2001.

[3] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer systems. Technical Report CS2002-0726, UCSD, Nov 2002.

[4] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proc. of IPTPS*, 2003.

[5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of HotOS*, 2003.

[6] W. J. Bolosky et al. Feasibility of a serverless distributed file system depoloyed on an existing set of desktop PCs. In *Proc. of SIGMETRICS*, 2000.

[7] B. Callaghan. *NFS Illustrated*. Addison Wesley, 1999.

[8] F. Dabek et al. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[9] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proc. of SRDS*, 2001.

[10] K. Keeton and J. Wilkes. Automating data dependability. In *Proc. of the ACM SIGOPS European Workshop*, 2002.

[11] D. Kostic et al. Using random subsets to build scalable services. In *Proc. of USITS*, 2003.

[12] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[13] P. Maymounkov and D. Mazieres. Rateless codes and big downloads. In *Proc. of IPTPS*, 2003.

[14] D. Mazieres. A toolkit for user-level file systems. In *Proc. of the USENIX technical conference*, 2001.

[15] A. Muthitacharoen et al. Ivy: a read-write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[16] Overnet website, http://www.overnet.com.

[17] D. A. Patterson et al. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB-CSD-02-1175, UC Berkeley, 2002.

[18] S. Saroiu et al. An Analysis of Internet Content Delivery Systems. In *Proc. of OSDI*, 2002.

[19] S. Saroiu et al. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN*, 2002.

[20] S. Savage and J. Wilkes. AFRAID – a frequently redundant array of independent disks. In *Proc. of the USENIX Technical Conference*, 1996.

[21] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM*, 2001.

[22] H. Weatherspoon et al. Silverback: A global-scale archival system. Technical Report UCB-CSD-01-1139, UC Berkeley, 2001.

[23] J. Wilkes et al. The HP AutoRAID hierarchical storage system. In *Proc. of SOSP*, 1995.

[24] J. J. Wylie et al. Survivable information storage systems. *IEEE Computer*, 2001.

TimeLine: A High Performance Archive for a Distributed Object Store

Chuang-Hue Moh and Barbara Liskov
MIT Computer Science and Artificial Intelligence Laboratory

Abstract

This paper describes TimeLine, an efficient archive service for a distributed storage system. TimeLine allows users to take snapshots on demand. The archive is stored online so that it is easily accessible to users. It enables "time travel" in which a user runs a computation on an earlier system state.

Archiving is challenging when storage is distributed. In particular, a key issue is how to provide consistent snapshots, yet avoid stopping user access to stored state while a snapshot is being taken. The paper defines the properties that an archive service ought to provide and describes an implementation approach that provides the desired properties yet is also efficient. TimeLine is designed to provide snapshots for a distributed persistent object store. However the properties and the implementation approach apply to file systems and databases as well.

TimeLine has been implemented and we present the results of experiments that evaluate its performance. The experiments show that computations in the past run well when the archive store is nearby, e.g., on the same LAN, or connected by a high speed link. The results also show that taking snapshots has negligible impact on the cost of concurrently running computations, regardless of where the archived data is stored.

1 INTRODUCTION

This paper describes TimeLine, an efficient archive service for a storage system. TimeLine allows users to take snapshots of system state on demand, either by issuing a command, and/or by running a program that requests snapshots periodically, e.g., once every six hours. The archive containing the snapshots is stored online so that it is easily accessible to users.

Access to the archive is based on time. Each snapshot is assigned a timestamp, reflecting the time at which it was requested. The archive enables "time travel" in which users can run computations "in the past." The user indicates the time at which the computation should run. The snapshot used to run the computation is the latest one whose timestamp is no greater than the specified time. We chose to use a time-based interface because we believe it matches user needs. Note however that users can easily build directory structures that allow them to associate names with snapshots.

TimeLine is designed to provide access to old states of objects in the Thor persistent object store (Thor [2, 13, 14]). Thor is a distributed system in which storage resides at many servers; it is designed to scale to very large size, with a large number of servers that are geographically distributed. Thor allows users to run computations that access the current states of objects; TimeLine extends this interface so that users can also run computations that access earlier states of objects.

The key issue in providing an archive is providing consistent snapshots without disrupting user access to stored state while a snapshot is being taken. This issue arises in any kind of storage system, not just a persistent object store. Consistency without disruption is easy to provide when all storage resides at a single server but challenging when storage is distributed.

To our knowledge, TimeLine is the first archive for a distributed storage system that provides consistent snapshots without disruption. Most other archive systems (e.g., [8, 24, 31]) disrupt user access to a greater or lesser extent while a snapshot is being taken. Elephant [26] does not cause such a disruption, but it is not a distributed system. In addition, Elephant snapshots all data; by taking snapshots on command we reduce archive storage and CPU costs associated with taking snapshots.

TimeLine has been implemented and we present the results of experiments that evaluate its performance. Our experiments show that taking snapshots has negligible impact on the cost of running computations that are using the "current" system state. We also present results showing the cost of using snapshots, i.e., running computations in the past. Our experiments show that computations in the past run as well as those in the present when the archive state is co-located with the current system state. However, it is not necessarily desirable to co-locate storage like this, since the archived state can become very large. Therefore we looked at a range of options for using shared storage for snapshots. Our results show that good performance for computations in the past is possible using shared archive storage provided it is reasonably close, e.g., on the same LAN, or connected by a high speed link. The results also show that the cost of running computations in the present while archiving is occurring is insensitive to where the archived data is stored.

The remainder of the paper is organized as follows. Section 2 discusses requirements for an archive system and describes our approach for providing consistency without disruption. Sections 3 to 6 describe TimeLine and its implementation. Section 7 presents our performance results and Section 8 discusses related work. We conclude in Section 9.

2 SNAPSHOT REQUIREMENTS

This section defines requirements for an archive service and describes our approach to satisfying them.

An archive must provide global consistency [18]:

- *Consistency.* A snapshot must record a state that could have existed at some moment in time.

Typically, an application controls the order of writes by completing one modification before starting another, i.e., by ensuring that one modification *happens before* another [11]. Then, a snapshot must not record a state in which the second modification has occurred but the first has not.

However, we cannot afford to disrupt activities of other users:

- *Non-Disruption.* Snapshot should have minimal impact on system access: users should be able to access the system in parallel with the taking of the snapshot and the performance of user interactions should be close to what can be achieved in a system without snapshots.

Note that non-disruption is typically not a requirement for a backup system, since backups tend to be scheduled for times when there is little user activity. In contrast, our system allows snapshots to be taken whenever users want, and if snapshotting isn't implemented properly, the impact on concurrent users can be unacceptable.

One way to get a consistent snapshot is to freeze the system. In a distributed system, freezing requires a two-phase protocol (such a protocol is used in [31]). In phase 1 each node is notified that a snapshot is underway. When a node receives this message, it suspends processing of user requests (and also notes that modifications that have happened up to that point will be in the snapshot but later ones will not). The coordinator of the protocol waits until it receives acks from all nodes. Then in phase 2 it notifies all nodes that the freeze can be lifted. Freezing provides consistency because it ensures that if a modification to x is not reflected in a snapshot, this modification will be delayed until phase 2, and therefore any modification that happened after it will also not be included in the snapshot. Freezing is necessary since otherwise x's node would not delay the modification to x and therefore the snapshot could observe some modification that happened after the modification of x, yet miss the modification of x.

Doing snapshots by freezing the system obviously does not satisfy our non-disruption goal. The approach is particularly problematical in a large-scale distributed system: when there are many nodes that may be widely distributed, the time required to run the protocol can be large. Also, if any storage node is down or not communicating, the freeze can last a very long time.

An alternative implementation is to record the information about the snapshot at one server, and have every server check this information before carrying out a modification. This approach is even less desirable than freezing, since it delays every modification.

We can avoid these problems if the system records information about happens before. This can be accomplished as follows: Each node includes its local time in every message it sends, and a node can perform a modification when the time of its own clock is later than any time it already heard of. (This protocol is a variation on one proposed in [17].) For example, if a user reads information from server X and then writes to server Y, information about server X's timestamp will flow to server Y, and server Y will delay performing the modification until its clock is larger than server X's clock was when it performed the modification to x. Note that the approach is cheap to implement and in practice modifications are unlikely to be delayed, provided server clocks are loosely synchronized.

Assuming timestamps as just described, we can implement snapshots as follows. A single server acts as the *snapshot coordinator*. To take a snapshot, a user communicates with this server. The server assigns a timestamp to the snapshot by reading its local clock. Information about the snapshot then flows to all servers, but this can be done in the background. The timestamp determines what modifications are contained in the snapshot: the snapshot includes modifications that occurred earlier than this time but not those that occurred after this time. To make this work, nodes need to track when modifications happen. This could be done by associating a "time-last-modified" timestamp with each object but in fact only information about recent modifications is needed, as discussed in Section 4.

The approach provides consistency without disruption. Taking a snapshot requires communication with the snapshot coordinator, which must be up and communicating. But such communication is also needed when all storage is at a single server and in either case, we can increase availability by using standard replication (e.g., [21]). There could be more than one snapshot coordinator; then snapshots would be ordered relative to modifications by using vector clocks [10, 23] with an entry for each coordinator.

With this approach it is possible that users might notice anomalies [6, 11]: a snapshot might fail to include a modification that a person knows happened before it, or it might include a modification that a person knows happened after it. Such anomalies require communication outside the system. E.g., if Bob takes a snapshot after being told by Alice that a certain modification has been made, the snapshot might nevertheless miss this modification. Anomalies are highly unlikely because current time synchronization technologies, such as NTP [20], synchronize clocks tightly enough to prevent them. Thus, the timestamp assigned to Bob's snapshot is highly likely to be greater than that of Alice's modification. (A similar use of loosely synchronized clocks to avoid such anomalies was proposed in [15].)

3 CREATING SNAPSHOTS

TimeLine provides snapshots for objects in Thor [13, 14]. This section provides a brief overview of Thor and then describes what happens when a user requests a snapshot.

3.1 Overview of Thor

Thor is a persistent object store based on the client-server model. Servers provide persistent storage for objects; we call these *object repositories (ORs)*. User code runs at client machines and interacts with Thor through client-side code called the *front end (FE)*. The FE contains cached copies of recently used objects.

A user interacts with Thor by running atomic transactions. An individual transaction can access (read and/or modify) many objects. A transaction terminates by committing or aborting. If it commits all modifications become persistent; if it aborts none of its changes are reflected in the persistent store.

Thor employs optimistic concurrency control to provide atomicity. The FE keeps track of the objects used by the current transaction. When the user requests to commit the transaction, the FE sends this information, along with the new states of modified objects, to one of the ORs. This OR decides whether a commit is possible; it runs a two-phase commit protocol if the transaction made use of objects at multiple ORs.

Thor uses timestamps as part of the commit protocol. Timestamps are globally unique: a timestamp is a pair, consisting of the time of the clock of the node that produced it, and the node's unique ID. Each transaction is assigned a timestamp, and can commit only if it can be serialized after all transactions with earlier timestamps and before all transactions with later timestamps.

Thor provides highly available and reliable storage for its objects, either by replicating their storage at multiple nodes using primary/backup replication, or by careful use of disk storage.

3.2 Taking a Snapshot

To create a snapshot, a user communicates with one of the ORs, which acts as the snapshot coordinator. This OR (which we will refer to as the SSC) assigns a timestamp to the snapshot by reading its local clock. The SSC serializes snapshot requests so that every snapshot has a unique timestamp, and it assigns later timestamps to later snapshots. It also maintains a complete *snapshot history*, which is a list of the assigned timestamps, and records this information persistently.

Once a snapshot request has been processed by the SSC, information about the snapshot must flow to all the ORs, and we would like this to happen quickly so that usually ORs have up-to-date information about snapshots. We could propagate snapshot information by having the SSC send the information to each OR, or we could have ORs request the information by frequent communication with the SSC, but these approaches impose a large load on the SSC if there are lots of ORs. Therefore, instead we propagate history information using gossip [3]: history information is piggybacked on every message in the system.

A problem with using gossip is that if the timestamp of the most recent snapshot in the piggybacked history is old, this might mean that no more recent snapshot has been taken, or it might mean that the piggybacked information is out of date. We distinguish these cases by having the SSC include its current time along with the snapshot history, thus indicating how recent the information is.

Gossip alone might not cause snapshot history information to propagate fast enough. To speed things up, we can superimpose a distribution tree on the ORs. To start the information flowing the SSC notifies a subset of ORs (those at the second level of the tree) each time it creates a new snapshot, and also periodically, e.g., every few seconds.

ORs can always fall back on querying the SSC (or one another) when snapshot information is not propagated fast enough. Even if the SSC is unreachable for a period of time, e.g., during a network partition, our design (see Section 4) ensures that the ORs still function correctly.

By using timestamps for snapshots, we are able to serialize snapshots with respect to user transactions: a snapshot reflects modifications of all user transactions with smaller timestamps. Serializing snapshots gives us atomicity, which is stronger than consistency. For example, we can guarantee that snapshots observe either all modifications of a user transaction, or none of them. However the approach of using timestamps also works in a system that provides atomicity for individual modifications but no way to group modifications into multi-operation transactions.

3.3 Snapshot Messages

Over time the snapshot history can become very large. However, usually only very recent history needs to be sent, because the recipient will have an almost up to date history already.

Therefore, our piggybacked snapshot messages contain only a portion of the snapshot history. In addition, a snapshot message contains two timestamps, TS_{curr} and TS_{prev}, that bound the information in the message: the message contains the timestamps of all snapshots that happened after TS_{prev} and before TS_{curr}.

An OR can accept a snapshot message m if $m.TS_{prev}$ is less than or equal to the TS_{curr} of the previous snapshot message that the OR accepted. Typically TS_{prev} is chosen by the sender to be small enough that the receiver is highly likely to be able to accept a message. Even with a TS_{prev} quite far in the past, the number of snapshot timestamps in the message is likely to be small. In fact, we expect a common case is that the message will contain no snapshot timestamps, since the rate of taking snapshots is low relative to the frequency at which history information is propagated.

If a node is unable to accept a snapshot message, it can request the missing information from another node, e.g., the sender.

4 STORING SNAPSHOTS

This section and the two that follow describe the main functions of TimeLine. This section decribes how the OR saves snapshot information in the archive. Section 5 describes the archive service, which stores snapshot information when requested to do so by ORs and also provides access to previously stored snapshots. Section 6 describes how TimeLine carries out user commands to run computations in the past.

4.1 Design Overview

TimeLine provides snapshots by using a combination of current pages at ORs and pages stored in the archive. In particular, if a page has not been modified since a snapshot occurred, then the copy on the OR disk contains the proper information. On the other hand, if the page has been modified, its previous state needs to be written to the archive before its disk copy is overwritten. Thus we use a copy-on-write scheme, specialized to work for snapshots.

To create a snapshot page the OR needs to know all snapshots whose timestamps are less than t, the maximum timestamp of any transaction whose modification is being recorded on disk by overwriting the page. A fundamental problem with a gossip scheme, however, is that an OR's history information is never completely up to date. Therefore it might not know about all snapshots whose timestamps are less than t. In particular, since gossip takes time to arrive, an OR won't know about timestamps of snapshots that were taken very recently.

We could solve this problem by creating snapshot pages just in case they are needed, but that is expensive, especially since most of the time they won't be needed. Instead, we would like to avoid unnecessary creation of snapshot pages. We accomplish this by delaying overwriting of pages on disk until we know whether snapshot pages are needed. Our approach is based on details of the OR implementation, which is discussed in the next section.

4.2 Thor ORs

This section describes relevant details of how Thor ORs are implemented.

Thor stores objects in 8KB pages. Typically objects are small and there are many of them in a page. Each page belongs to a particular OR. An object is identified uniquely by the 32-bit ORnum of its OR and a 32-bit OREF that is unique within its OR. The OREF contains the PageID of the object's page and the object's offset within the page.

The FE responds to a cache miss by fetching an entire page from the OR, but only modified objects are shipped back from a FE to an OR when the transaction commits. This means that in order to write the modifications to the page on disk, the OR will usually need to do an *installation read* to obtain the object's page from disk. If we had to do this read as part of committing a transaction, it would degrade system performance.

Therefore instead the OR uses a volatile *Modified Object Buffer* (MOB) [5] to store modifications when a transaction commits, rather than writing them to disk immediately. This approach allows disk reads and writes to be deferred to a convenient time and also enables *write absorption*, i.e., the accumulation of multiple modifications to a page so that these modifications can be written to the page in a single disk write. Use of the MOB thus reduces disk activity and improves system performance.

Using the MOB does not compromise the correctness of the system because correctness is guaranteed by the transaction log: modifications are written to the transaction log and made persistent before a transaction is committed.

Entries are removed from the MOB when the respective modifications are installed in their pages and the pages are written back to disk. Removal of MOB entries (*cleaning* the MOB) is done by a *flusher thread* that runs in the background. This thread starts to run when the MOB fills beyond a pre-determined *high watermark* and removes entries until the MOB becomes small enough. The flusher processes the MOB in log order to facilitate the truncation of the transaction log. For each modification it encounters, it reads the modified object's page from disk, installs all modifications in the MOB that are for objects in that page, and then writes the updated page back to disk. When the flusher finishes a pass of cleaning the MOB it also removes entries for all transactions that have been completely processed from the transaction log.

The OR also has a volatile page buffer that it uses to satisfy FE fetch requests. Before returning a page to the requesting FE, the OR updates the buffer copy to contain all modifications in the MOB for that page. This ensures that pages fetched from the OR always reflect committed transactions. Pages are flushed from the page buffer as needed (LRU). Dirty pages are simply discarded rather than being written back to disk; the flusher thread is therefore the only part of the system that modifies pages on disk.

4.3 The Anti-MOB

Our implementation takes advantage of the MOB to delay the writing of snapshot pages until the snapshot history is sufficiently up to date.

To know whether to create snapshot pages, an OR needs to know the current snapshot status for each of its pages. This information is kept in its *snapshot page map*, which stores a timestamp for each page. When a snapshot page for page p is created due to snapshot S, the corresponding snapshot page map entry is updated with the timestamp of S. All snapshot page map entries are initially zero. Snapshots of a page are created in timestamp order; therefore if the snapshot page map entry for a page has a timestamp greater than that of a snapshot, the OR must have already created a snapshot copy of the page for that snapshot.

An OR also stores what it knows of the snapshot history together with T_{gmax}, the highest SSC timestamp it has received.

Recall that the only time pages are overwritten on disk is when the MOB is cleaned. Therefore we can also create snapshot pages as part of cleaning the MOB. Doing so has two important advantages. First, it allows us to benefit from the installation read already being done by the flusher thread. Second, because the flusher runs when the MOB is almost full, it works on transactions that committed in the past. Thus, by the time a transaction's modifications are processed by the flusher, the OR is highly likely to have snapshot information that is recent enough to know whether snapshot pages are required.

A simple strategy is to start with the page read by the flusher thread and then use the MOB to obtain the right state for the snapshot page. For example, suppose the MOB contains modifications for two transactions T_1 and T_2, both of which modified the page, and suppose also that the snapshot for which the snapshot page is being created has a timestamp that is greater than T_1 but less than T_2. Then we need to apply T_1's modifications to the page before writing it to the archive.

However, this simple strategy doesn't handle all the situations that can arise, for three reasons. First, if the page being processed by the flusher thread is already in the page buffer, the flusher does not read it from disk. But in this case, the page already reflects modifications in the MOB, including those for transactions later than the snapshot. We could get the pre-states for these modifications by re-reading the page from disk, but that is undesirable. Hence, we need a way to revert modifications.

A second problem is that if a transaction modifies an object that already has an entry in the MOB (due to an older transaction), the old MOB entry is overwritten by the new one. However, we may need the overwritten entry to create the snapshot page.

Finally, if the MOB is processed to the end, we will encounter modifications that belong to transactions committed after T_{gmax}. Normally we avoid doing this: the flusher does not process MOB entries recording modifications of transactions with timestamps greater than T_{gmax}. But sometime the flusher must process such entries – when propagation of the snapshot history is stalled. When this happens we must be able to continue processing the MOB since otherwise the OR will be unable to continue committing transactions. But to do so, we need a place to store information about the

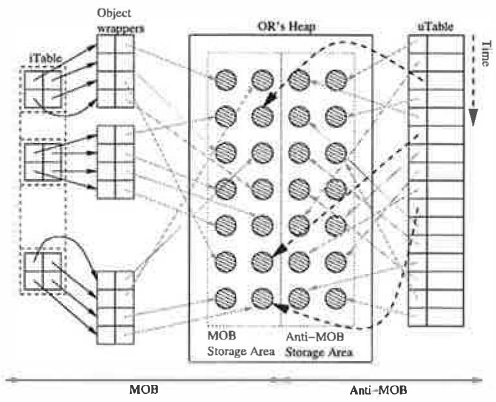


Figure 1. Structure of the Anti-MOB

pre-image of the page before it is overwritten on disk.

The OR uses an in-memory buffer called the *Anti-MOB* to store overwritten MOB entries and transaction pre-images. The Anti-MOB (like the MOB) records information about objects. As shown in Figure 1, each Anti-MOB entry contains a pointer to the object's pre-image in the OR's heap, which it shares with the MOB. The entry also contains the timestamp of the pre-image's transaction, i.e., the transaction that caused that pre-image to be stored in the Anti-MOB. For example, if T_2 overwrites an object already written by T_1, a pointer to the older version of the object is stored in the Anti-MOB along with T_2's timestamp. Note that we don't make a copy of the old version; instead we just move the pointer to it from the MOB to the Anti-MOB.

Information is written to the Anti-MOB only when it is needed or might be needed. The Anti-MOB is usually needed when an object is overwritten, since the committing transaction is almost certain to have a timestamp larger than T_{gmax}. But when a page is read from disk in response to a fetch request from an FE, a modification installed in it might be due to a transaction with an old enough timestamp that the OR knows the pre-state will not be needed.

In addition, the first time an object is overwritten, we may create an additional Anti-MOB entry for the transaction that did the original write. This entry, which has its pointer to the pre-image set to null, indicates that the pre-image for that transaction is in the page on disk. For example, suppose transaction T_1 modified object x and later transaction T_2 modifies x. When the OR adds T_2's modified version of x to the MOB, it also stores $< T_2, x, x_1 >$ in the Anti-MOB to indicate that x_1 (the value of x created by T_1) is the pre-image of x for T_2. Furthermore, if T_1's modification to x was the first modification to that object by any transaction currently recorded in the MOB and if there is or might be a snapshot S before T_1 that requires a snapshot page for x's page, the OR also adds $< T_1, x, null >$ to the Anti-MOB, indicating that for S, the proper value of x is the one currently stored on disk.

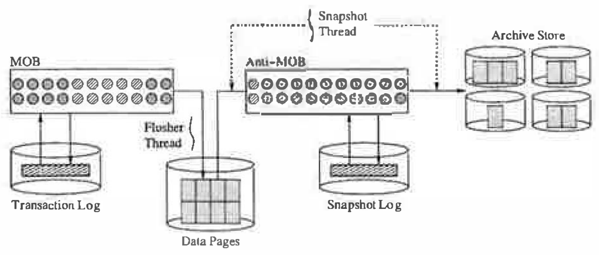


Figure 2. The Snapshot Subsystem in an OR

```
Let SCAN_(P,S) = set of objects scanned for page P
for snapshot S and initialized to ∅

for each entry U_i in the Anti-MOB for a transaction
with timestamp ≥ S's timestamp, S.t
    if U_i.Oref ∉ SCAN_(P,S) then
        SCAN_(P,S) = SCAN_(P,S) ∪ {U_i.Oref}
        install U_i into P
    endif
end
```

Figure 3. Algorithm for Creating Snapshot Pages

4.3.1 Creating Snapshot Pages

Now we describe how snapshot pages are created. Figure 2 shows the design of the snapshot subsystem, which works in tandem with the OR's MOB and transaction log. The work of taking snapshots is done partly by the flusher thread and partly by the *snapshot thread*, which also works in the background.

Snapshot pages are produced using both the MOB (to apply modifications) and the Anti-MOB (to revert modifications). As mentioned, to install a modification in the MOB, the flusher thread reads the page from disk if necessary. Then, it goes through the MOB and applies each modification for that page to the in-memory copy provided the transaction that made the modification has a timestamp less than some bound. Typically this bound will be T_{gmax}, but occasionally (when the history information is old), it will be larger than this.

Before applying a modification to the page, the flusher decides whether a pre-image is or might be needed. The pre-image might be needed if the transaction that did the modification has a timestamp greater than T_{gmax}; it will be needed if the OR knows of a snapshot with timestamp less than that of the transaction, and where the entry for the page in the snapshot page map is less than that of the snapshot.

To make a pre-image the flusher copies the current value of the modified object, x, from the page into the heap before overwriting the object. If there is no entry in the Anti-MOB for x and T, the transaction that caused that modification, it creates one; this entry points to x's value in the heap. Otherwise, if there is an Anti-MOB entry for x containing null, the flusher modifies the entry so that it now points to x's value in the heap.

If the page being modified is already in the page cache, the flusher applies the modifications to this copy. If this copy already contains modifications, the appropriate pre-images will exist in the Anti-MOB because they were put there as part of carrying out the fetch request for that page.

Once the flusher has applied the modifications to the page, and assuming a snapshot is required for the page, it makes a copy of the page and then uses the Anti-MOB to revert the page to the proper state. To undo modifications to page P as needed for a snapshot S with timestamp S.t, the OR scans the portion of the Anti-MOB with timestamps greater than S.t for pre-images corresponding to modifications to P and installs them in P. The algorithm used for creating snapshot pages is shown in Figure 3. The Anti-MOB can contain several pre-images corresponding to a single object. However, only the pre-image of the oldest transaction more recent than a snapshot will contain the correct version of the object for that snapshot. The algorithm uses the SCAN set to keep track of pre-images scanned and ensure that only the correct pre-image corresponding to an object (i.e., the first one encountered) is used for creating the snapshot page.

A snapshot page belongs to the most recent snapshot, S, with timestamp less than that of the earliest transaction, T, whose pre-image is placed in the page. The OR can only know what snapshot this is, however, if T has a timestamp less than T_{gmax}. Therefore snapshot pages are produced only when this condition holds. Section 4.5 discusses what happens when the OR cleans the MOB beyond T_{gmax}.

The snapshot page must be made persistent before the disk copy of the page can be overwritten. We discuss this point further in Section 4.4.

4.3.2 Discarding Anti-MOB Entries

Entries can be discarded from the Anti-MOB as soon as they are no longer needed for making snapshot pages.

The flusher makes snapshot pages up to the most recent snapshot it knows about for all pages it modifies as part of a pass of cleaning the MOB. At the end of this pass all entries used to make these snapshot pages can be deleted from the Anti-MOB. Such entries will have a timestamp that is less than or equal to T_{gmax} (since all snapshot pages for P for snapshots before T_{gmax} will have been made by that point).

When new snapshot information arrives at an OR, the OR updates T_{gmax}. Then it can re-evaluate whether Anti-MOB entries can be discarded. More specifically, when the OR advances its T_{gmax} from t_1 to t_2, it scans its Anti-MOB for entries with timestamps between t_1 and t_2. An entry can be discarded if the corresponding snapshot page of the most recent snapshot with a timestamp smaller than itself has already been created. Usually, all such entries will be removed (because there is no snapshot between t_1 and t_2).

4.4 Persistence and Failure Recovery

Once a page has been modified on disk, the Anti-MOB is the only place where pre-images exist until the snapshot pages that require those pre-images have been written to the archive. Yet the Anti-MOB is volatile and would be lost if the OR failed.

Normally snapshot pages are created as part of cleaning the MOB, and we could avoid the need to make information in the Anti-MOB persistent by delaying the overwrite of a page on disk until all its snapshot pages have been written to the archive. However, we decided not to implement things this way because the write to the archive might be slow (if the snapshot page is stored at a node that is far away in network distance). Also, there are cases where we need to overwrite the page before making snapshot pages, e.g., when propagation of snapshot history stalls (see Section 4.5).

Therefore we decided to make use of a snapshot log as a way of ensuring persistence of pre-images. Pre-images are inserted into the snapshot log as they are used for reverting pages to get the appropriate snapshot page. Thus, the creation of a snapshot page causes a number of entries for that page to be added to the snapshot log. Together, these entries record all pre-images for objects in that page whose modifications need to be reverted to recover the snapshot page.

Before a page is overwritten on disk, snapshot log entries containing the pre-images of modifications to the page are flushed to disk (or to the OR backups). The snapshot page can then be written to the archive asynchronously. Note that we can write snapshot log entries to disk in a large group (covering many snapshot pages) and thus amortize the cost of the flush.

Entries can be removed from the snapshot log once the associated snapshot pages have been saved to the archive.

4.4.1 Recovery of Snapshot Information

When the OR recovers from a failure, it recovers its snapshot history by communicating with another OR, initializes the MOB and Anti-MOB to empty, and initializes the snapshot page map to have all entries "undefined". Then it reads the transaction log and places information about modified objects in the MOB. As it does so, it may create some Anti-MOB entries if objects are overwritten. Then it reads the snapshot log and adds its entries to the Anti-MOB. The snapshot log is processed by working backwards. For example, suppose P required two snapshot pages P_{S_1} and P_{S_2} for snapshots S_1 and S_2 respectively ($S_1.t < S_2.t$). To recover P_{S_1}, the system reads P from disk, applies the entries in the snapshot log for P_{S_2}, and then applies the entries for P_{S_1}.

At this point, the OR can resume normal processing, even though it doesn't yet have correct information in the snapshot page map. The OR obtains this information from the archive. For example, when it adds an entry to the MOB, if the entry for that page in the snapshot page map is undefined, the OR requests the snapshot information from the archive. This is done asynchronously, but by the time the information is needed for cleaning the MOB, the OR is highly likely to know it.

4.5 Anti-MOB Overflow

When propagation of snapshot history information stalls, the OR must clean the MOB beyond T_{gmax} and this can cause the Anti-MOB to overflow the in-memory space allocated to it. Therefore at the end of such a cleaning, we push the current contents of the Anti-MOB into the snapshot log and force the log to disk. Then we can clear the Anti-MOB, so that the OR can continue inserting new entries into it.

When the OR gets updated snapshot information and advances its T_{gmax}, it processes the Anti-MOB blocks previous written to the snapshot log. It does this processing backward, from the most recently written block to the earliest block. Each block is also processed backward. Of course, if the OR learns that no new snapshots have been taken, it can avoid this processing and just discard all the blocks. It can also stop its backward processing as soon as it reaches a block all of whose entries are for transactions that are earlier than the earliest new snapshot it just heard about.

To process a block, the OR must use the appropriate page image. The first time it encounters an entry for a particular page, this will be the current page on disk. But later, it needs to use the page that it already modified. Therefore the OR uses a table that records information about all pages processed so far. These pages will be stored in memory if possible, and otherwise on disk.

Once snapshot pages have been written to the archive, the OR removes the Anti-MOB blocks from the snapshot log and discards all the temporary pages used for processing the blocks.

5 ARCHIVING SNAPSHOTS

This section describes the design of the archive store. We require that the archive provide storage at least as reliable and available as what Thor provides for the current states of objects. For example, if Thor ORs are replicated, snapshot pages should have the same degree of replication.

5.1 Archive Store Abstraction

Table 1 shows the interface to the archive. It provides three operations: put, get, and getTS.

The put operation is used to store a page in the archive. Its arguments provide the value of the snapshot page and its identity (by giving the ORnum of its OR, its PageID within the OR, and the timestamp of the snapshot for which it was created). The operation returns when the page has been

Operation	Description
put(ORnum, PageID, TS_{ss}, $Page_{ss}$)	Stores the snapshot page $Page_{ss}$ into the archive
get(ORnum, PageID, TS_{ss})	Retrieves the snapshot page from the archive.
getTS(ORnum, PageID)	Retrieves the latest timestamp for the page.

Table 1. Interface to the Archive

stored reliably, e.g., written to disk, or written to a sufficient number of replicas.

The get operation is used to retrieve a page from the archive. Its arguments identify the page by giving its ORnum and PageID, and the timestamp of the snapshot of interest. If the archive contains a page with that identity and timestamp, it returns it. Otherwise, if there is a later snapshot page for that page, it returns the oldest one of these. Otherwise it returns null.

It is correct to return a later snapshot page because of the way we produce snapshot pages: we only produce them as pre-images before later modifications. If there is no snapshot page with the required timestamp but there is a later one, this means there were no changes to that page between the requested snapshot and the earliest later snapshot, and therefore the information stored for the later snapshot is also correct for this snapshot.

The getTS operation takes an ORnum and PageID as arguments. It returns the latest timestamp of a snapshot page for the identified page; it returns zero if there is no snapshot for that page. It is used by the OR to reinitialize its snapshot page map when it recovers from a failure.

5.2 Archive Store Implementations

We considered three alternatives for implementing the archive store: a local archive store at each OR, a network archive store on the local area network, and an archive store on servers distributed across a wide area network.

A local archive store archives the snapshot pages created at its OR. As we will discuss further in Section 6, all snapshot pages are fetched via the OR. Therefore this approach will provide the best performance (for transactions on snapshots) as compared to the other two implementations. On the other hand, the archive can potentially become very large. In addition, ORs may differ widely in how much archive space they require, so that it may be desirable to allow sharing of the archive storage.

In a network archive store, snapshots are archived in a specialized storage node located close to a group of ORs, e.g., on the same LAN. This design allows sharing of what might be a high-end storage node or nodes, e.g., a node with a disk array that provides high reliability. The impact of a network archive store on the performance of fetching snap-

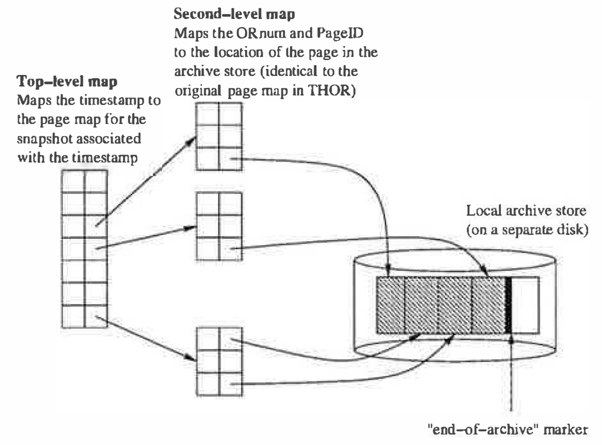

Second–level map
Maps the ORnum and PageID to the location of the page in the archive store (identical to the original page map in THOR)
Top–level map
Maps the timestamp to the page map for the snapshot associated with the timestamp
Local archive store (on a separate disk)
"end–of–archive" marker

Figure 4. Storage System of an Archive Store Node

shot pages depends on the speed of the underlying network connection. For example, on a gigabit LAN, we expect the performance of the network archive store to be close to that of a local archive store.

A distributed archive store is implemented as a set of storage nodes that are distributed across the wide area network. All ORs share the archive store and therefore this approach allows even more sharing than the network archive store. Also, with proper design, the system can provide excellent reliability, automatic load-balancing, and self-organization features, similar to what can be achieved, e.g., in peer-to-peer systems [25, 28]. The downside of this scheme is that the data may be located far away from the ORs, making it costly to run transactions on snapshots.

We expect the choice of where to place the archive store to mainly affect the cost of running computations on snapshots. All designs should perform similarly with respect to taking snapshots since writing snapshot pages to the archive is done asynchronously.

5.3 Implementation Details

In this section, we describe the archive store implementations.

In a local archive store, snapshots are stored on a separate disk (or disks) so that requests to read or write snapshot pages don't interfere with normal OR activity, e.g., fetching current pages.

The disk is organized as a log as shown in Figure 4. Snapshots pages are appended to the log, along with their identifier (ORnum, PageID, and timestamp of the snapshot for which the snapshot page was created). This organization makes writing of snapshot pages to disk fast (and we could speed it up even more by writing multiple pages to the log at once). A directory structure is maintained in primary memory to provide fast access to snapshot pages. The identifier information in the log allows the OR to recover the directory from the log after a failure.

A network archive store uses the same log-structured organization for its disk. The OR runs a client proxy that receives calls of archive operations such as put, turns them into messages, and communicates with the storage server using UDP.

Each storage node in a distributed archive store also uses the same log-structured organization. These nodes may be ORs, separate storage nodes, or a combination of the two.

Again, each OR runs a client proxy that handles calls of archive operations. But now the proxy has something interesting to do.

The main issue in designing a distributed archive store is deciding how to allocate snapshot pages to storage nodes. We decided to base our design on consistent hashing [9]. With this approach each storage node in the (distributed) archive store is assigned an identifier, its *nodeID*, in a large (160 bit) space, organized in a ring. The nodeIDs are chosen so that the space is evenly populated. Each snapshot page also has an identifier, its *archiveID*. To map the snapshot page to a storage node, we use the *successor* function as in Chord [28]; the node with the nodeID that succeeds the snapshot page's archiveID will be responsible for archiving that page. However, note that any deterministic function will work, e.g., we could have chosen the closest node in the ID space, as in Pastry [25].

The archiveID of a snapshot page is the SHA1 hash of the page's ORnum and PageID. This way, all snapshot pages of the same page will be mapped to the same storage node. This choice makes the implementation of get and getTS efficient because they can be implemented at a single node.

To interact with the archive, the client proxy at the ORs needs to map the NodeID of the node responsible for the page of interest to its IP address. Systems based on consistent hashing typically use a multi-step routing algorithm [25, 28] to do this mapping although recent work shows that routing can be done in one step [7]. However, routing is not an important concern for us because the client proxy at an OR can cache the IP addresses of the nodes responsible for its OR's snapshot pages. Since the population of storage nodes is unlikely to change very frequently, the cache hit rate will be high. Therefore, the cost of using distributed storage will just be the cost of sending and receiving the data from its storage node.

We chose to use consistent hashing for a number of reasons. If the node IDs and snapshot IDs are distributed reasonably, it provides good load balancing. It also has the property of allowing nodes to join and leave the system with only local disruption: data needs to be redistributed when this happens but only nodes nearby in ID space are affected.

Of course, snapshot pages need to be replicated to allow storage nodes to leave the system without losing any data. Thus a snapshot page will need to be archived at several nodes. This does not slow down the system but does generate more network traffic. One benefit of replication is that snapshot pages can be retrieved in parallel from all the replicas and the first copy arriving at the OR can be used to satisfy the request. As a result, the cost of retrieving a snapshot page is the cost of fetching it from the nearest replica.

6 TRANSACTIONS ON SNAPSHOTS

Users can run transactions on a snapshot by specifying a time in the "past" at which the transaction should execute. We only allow read-only transactions, since we don't want transactions that run in the past to be able to rewrite history.

The most recent snapshot, S, with timestamp less than or equal to the specified timestamp will be the snapshot used for the transaction. The FE uses the timestamp of S to fetch snapshot pages. Since the FE needs to determine S, the timestamp specified by the user must be less than the T_{gmax} at the FE.

In this section, we first provide an overview of the FE. Then we describe how the FE and OR together support read-only transactions on snapshots.

6.1 Thor Front-End

This section contains a brief overview of the FE. More information can be found in [2, 13, 14].

To speed up client transactions, the FE maintains copies of persistent objects fetched from the OR in its in-memory *cache*. It uses a *page map* to locate pages in the cache, using the page's PageID and the ORnum. When a transaction uses an object that isn't already in the cache, the FE fetches that object's page from its OR. When a transaction commits, only modified objects are shipped back to the OR.

Pages fetched from the OR are stored in page-size page frames. However when a page is evicted from the cache (to make room for an incoming page), its hot objects are retained by moving them into another *compacted frame* that stores such objects.

Objects at ORs refer to one another using OREFs (recall that an OREF identifies an object within its OR). To avoid the FE having to translate an OREF to a memory location each time it follows a pointer, we do *pointer swizzling*: the first time an OREF is used, it is replaced by information that allows the object to be efficiently located in the FE cache. When an OREF is swizzled, it is changed to point to an entry in the *Resident Object Table* (ROT). This entry then points to the object if it is in the cache. This way finding an object in the cache is cheap, yet it is also cheap to discard pages from the cache and to move objects into compacted pages.

6.2 Using Snapshots

When a user requests to run a transaction in the past, the FE cache is likely to contain pages belonging to other time lines, e.g., to the present. Similarly, when the user switches back from running in the past to running in the present, the FE cache will contain snapshot pages. In either case, the FE must ensure that the user transaction uses the appropriate objects: objects from current pages when running in the present; objects from snapshot pages for the requested snapshot when running in the past.

One approach to ensuring that the right objects are used is to clear the FE's cache each time the user switches to running at a different time. However, this approach forces the FE to discard hot objects from its cache and might degrade transaction performance. This could happen if the user switches repeatedly, e.g., from the current database to a snapshot and back to the current database again.

Therefore we chose a different approach. We extended the FE's cache management scheme so that the FE caches pages of multiple snapshots at the same time. Yet the FE manages its cache so that a transaction only sees objects of the correct version. Our implementation is optimized to have little impact on the running of transactions in the present, since we expect this to be the common case.

When the user switches from one time line to another, all entries in the ROT will refer to objects belonging to the time line it was using previously. We must ensure that the transaction that is about to run does not use these objects. We accomplish this by setting every entry in the ROT to null; this, in turn, causes ROT "misses" on subsequent accesses to objects.

When a ROT miss happens, the FE performs a lookup on the page map to locate the page on the cache. To ensure that we find the right page, one belonging to the time line being used by the currently running transaction, we extend the page map to store the timestamp for each page; current pages have a null timestamp. The page map lookup can then find the right page by using the timestamp associated with the currently running transaction. The lookup will succeed if appropriate page is in the cache. Otherwise, the lookup fails and the FE must fetch the page.

6.3 Fetching Snapshot Pages

The FE fetches a snapshot page by requesting it from the OR that stores that page. The reason for using the OR is that snapshots consist of both current pages and pages in the archive; the OR can provide the current page if that is appropriate, and otherwise it fetches the required page from the archive.

Page fetch requests from the FEs to the ORs are extended to contain a snapshot timestamp, or null if the FE transaction is running in the present. If the timestamp is not null, the OR uses it to determine the correct snapshot page by consulting its snapshot page map.

The snapshot page map stores the most recent snapshot timestamp for each page. If the timestamp of the request is less than or equal to the timestamp stored in the map for the requested page, the OR requests the snapshot page from the archive and returns the result of the fetch to the FE. Otherwise, the OR must create the snapshot page at that moment. It does this by using the page copy on disk or in the page buffer, plus the information in the MOB and Anti-MOB. It returns the resulting page to the FE. In addition if the snapshot page is different from the current page, the OR stores the page in the archive and updates its snapshot page map; this avoids creating the snapshot page again.

7 EXPERIMENTS

In this section, we present a performance evaluation of our snapshot service. We do this by comparing the performance of the original Thor to that of *Thor-SS*, the version of Thor that supports snapshots. Thor is a good basis for this study because it performs well. Earlier studies showed that it delivers comparable performance to a highly-optimized persistent object system implemented in C++ even though the C++ system did not support transactions [14]. Our experiments show that Thor-SS is only slightly more costly to run than Thor.

Thor-SS is a complete implementation of our design in the absence of failures; it includes making the Anti-MOB persistent but not the recovery mechanism. This implementation is sufficient to allow us to measure performance in the common case of no failures.

Our experimental methodology is based on the single-user OO7 benchmark [1]; this benchmark is intended to capture the characteristics of various CAD applications. The OO7 database contains a tree of *assembly* objects with leaves pointing to three *composite* parts chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by bidirectional *connection* objects, reachable from a single *root* atomic part; each atomic part has three connections. We employed the medium OO7 database configuration, where each composite part contains 200 atomic parts. The entire database consumes approximately 44 MB.

We used OO7 traversals T1 and T2B for our experiments. T1 is read-only and measures the raw traversal speed by performing a depth-first search of the composite part graph, touching every atomic part. T2B is read-write; it updates every atomic part per composite part.

We used a single FE and a single OR for our experiments. The OR and FE ran on separate Dell Precision 410 workstations (Pentium III 600 Mhz, 512 MB RAM, Quantum Atlas 10K 18WLS SCSI hard disk) with Linux kernel 2.4.7-10. Another identical machine is used for the network storage node in the network archive store, as well as for running a simulator (developed using the SFS toolkit [19]) for simulating a 60 node wide-area archive store. The machines were connected by a 100 Mbps switched Ethernet.

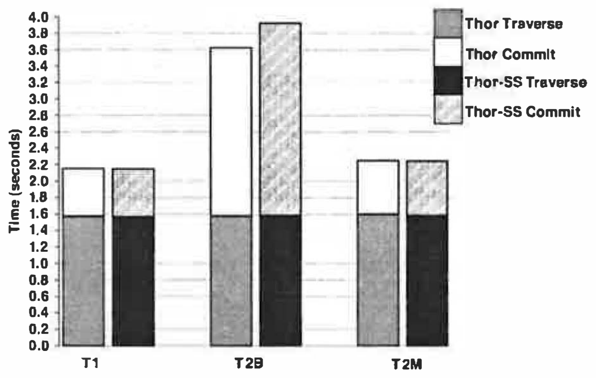


Figure 5. Baseline Comparison - Thor vs. Thor-SS

7.1 Foreground Costs

The first set of experiments compares the foreground cost of running Thor with Thor-SS. Particularly, we looked at the extra cost of committing transactions in Thor-SS when no MOB cleaning is occurring. The experiments used a 32 MB MOB to ensure MOB cleaning does not happen.

Thor-SS has extra work to do when the committing transaction overwrites previously modified objects. When overwriting occurs in Thor, the OR writes the modified objects to the MOB and modifies the MOB table to point to the new entry. This work also occurs in Thor-SS, but in addition Thor-SS must add the previous entry to the Anti-MOB. In either case this work is in the foreground: the OR doesn't respond to the FE's commit request until after work is done.

We measured the average execution times for T1 and T2B. In each case, the traversal was executed 20 times and each run was a separate transaction. The experiments were done with a hot FE cache that contained all the objects used in the traversal. This way we eliminated the cost of fetching pages from the OR, since otherwise the cost of fetching pages would dominate the execution time. The number of pages used in the traversals was identical in both Thor and Thor-SS.

The results of these experiments are shown in Figure 5. The figure shows that there is no significant difference between running the read-only traversal T1 in Thor and Thor-SS: in this case no writing to either the MOB or Anti-MOB occurs. In T2B, however, 98,600 objects were modified, which causes 98,600 overwrites in the OR's MOB and each overwrite leads to an object being moved to the Anti-MOB, leading to an 8% slowdown.

The CPU time for the OR to commit a T2B transaction is 1.65 seconds in Thor and 1.89 seconds in Thor-SS. The difference is the time taken by Thor-SS to do the extra work of storing overwritten objects into the Anti-MOB; the data show that it takes approximately 2.4 μsecs to update the Anti-MOB for each overwritten object.

The workload generated by T2B is not a very realistic representation of what we expect users to do: users are unlikely to repeatedly modify the same objects or modify almost all objects in the database. Therefore, we developed another workload that is more realistic. This workload, T2M, is a modified version of T2B: each T2M traversal randomly updates 10% (or 9,860 objects per traversal) of the objects that the original T2B modifies. In T2M there is much less overwriting than in T2B: each traversal overwrites an average of 3,852 objects. As shown in Figure 5, there is little difference in performance between Thor and Thor-SS for this traversal.

7.2 Impact of Snapshot Page Creation

In this section we look at the additional work that Thor-SS has to do when the MOB is cleaned. Thor cleans the MOB in the background using a low-priority process (the flusher) running in small time slices. Therefore, unless the OR is very heavily loaded, cleaning does not show up as a user-observable slowdown. We would like the same effect in Thor-SS, even when snapshot pages are being created.

Thor cleans several pages at once. To amortize disk seek time, it works on segments, which are 32 KB contiguous regions of disk, each containing four pages. Thor can read or write a segment about as cheaply as reading or writing a page, and if more than one page in the segment has been modified, cleaning costs will be lower using this approach.

The results in this section were obtained by running T2M (the modified T2B). In each experiment T2M ran 25 times, i.e., we committed 25 transactions. We ran experiments both with and without cleaning. To avoid cleaning we used an 8 MB MOB; to cause cleaning to occur we used a 4 MB MOB. With the 4 MB MOB, cleaning begins when the 12th traversal commits. During the remaining traversals, 1,141 segments were cleaned and 3,097 pages were modified; an average of 56 modified objects (consuming 2.6 KB) were installed in each segment. We continued to use a 32 MB cache at the OR so that no disk activity occurred during the cleaning part of the experiment.

To get a sense of the OR load due to cleaning, we measured the OR CPU time required to do cleaning. The results of these experiments are shown in Table 2. In each case the measurement started at the point where the flusher thread decided that a segment needs to be updated. At that point the segment is already in memory. The measurement for Thor stops when all modifications have been installed in the segment and their entries removed from the MOB, but the segment has not yet been written to disk. The results show the average cost of cleaning a segment.

The table shows two measurements for Thor-SS. The first is for the case where no snapshots need to be made; however, Thor-SS still has extra work to do, since entries need to be removed from the Anti-MOB. The second measurement is for Thor-SS when snapshot pages are being made. In this case the measurement includes the cost of creating snapshot pages, reverting the changes, and writing the pre-images to the snapshot log; the measurement stops

System	Cleaning Time
Thor	6.74 ms
Thor-SS without snapshot	6.87 ms
Thor-SS with snapshot	11.09 ms

Table 2. Time Taken to Clean a Segment

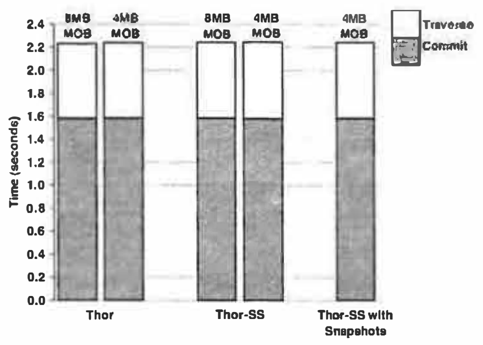


Figure 6. Cost of Taking Snapshots

before the snapshot log is forced to disk (since the flusher does not wait for this to complete). This is a worst case experiment in which every page requires a corresponding snapshot page, e.g., it represents a case where a snapshot has just been requested.

The impact on users of the additional cleaning activity in Thor-SS depends on the load at the OR. What we can hope is that when the OR has some spare cycles, we can run Thor-SS with cleaning and snapshot pages entirely in the background, just like Thor cleaning can be done in the background in this case. This is a reasonable expectation since, as we showed in Table 2, Thor-SS requires a modest amount of time to do cleaning even in the case where every page requires a snapshot page.

Figure 6 shows the results of these experiments. For both Thor and Thor-SS, the experiments compare the cost of running traversals with and without cleaning. The performance of the traversals is identical from the viewpoint of the user transaction: cleaning the MOB does not cause any slowdown. In particular, even in the case where every page that is cleaned requires a snapshot page, there is no slowdown.

7.3 Running Transactions on Snapshots

In this section, we examine the performance of running transactions in the past. We also compare this performance with that of running transactions in the present. These experiments use traversal T1, since we only allow read-only transactions to run in the past.

We ran T1 using a cold FE cache so that every page used by T1 must be fetched. Running T1 causes 5,746 snapshot pages to be fetched. We also used an empty page cache at the OR. Using a cold FE cache and an empty OR page cache provide us with a uniform setup to measure the slowdown that is caused by running T1 on a snapshot as compared to

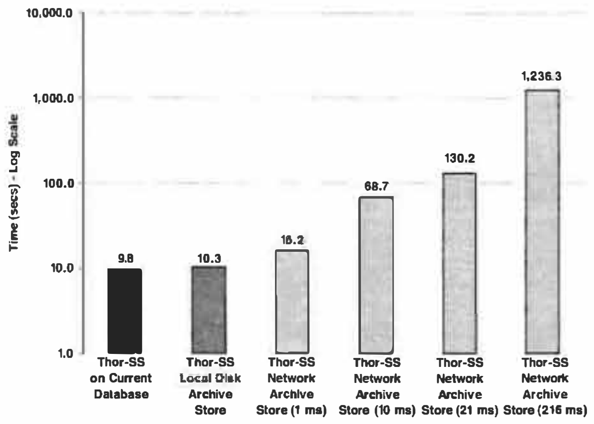


Figure 7. T1 Performance on Snapshot

running it on the current database. As discussed in Section 5, whenever there is a miss in the FE cache, the FE fetches the missing page from its OR. This is true whether the FE requires a snapshot page or a current page.

Figure 7 shows the results. The figure shows that running in the past is only slightly slower (approximately 5% slower) than running in the present when snapshots are stored on the OR's disk. This slowdown is due entirely to processing at the FE: in particular, it is due to an additional level of indirection in the FE's page map that affects transactions in the past but not in the present.

The figure also shows what happens when the archive store is not located at the OR. The slowdown is relatively small if the snapshot pages are archived at a nearby node that is connected to the OR by a 100 Mbps network, so that the cost to send 8 KB of data from the storage node to the OR is 1 ms: in this case the time taken to run T1 is 16.2 secs. Performance is more degraded when the the time required to send 8 KB of data increases to 10 ms, such as on a heavily loaded or slower (e.g., 10 Mbps) LAN; hence the performance of T1 is slower by an order of magnitude (68.7 secs). Performance is further degraded when the archive is even farther away.

These results imply that if the remote storage node were very close to the OR, e.g., connected to it by a gigabit network, performance of running on a snapshot would be essentially identical when storing the archive at the OR or at the remote node. Performance of running on snapshots degrades substantially as the storage moves farther away, but use of remote shared storage may still be the best choice because of the advantage of using such a shared facility.

8 RELATED WORK

A number of systems have provided snapshots, either for use in recovering from failure, or to provide the ability to look at past state. There is also a large body of work on systems that take checkpoints [4, 16]. Here we compare TimeLine to systems that provide snapshots, since they are most closely related.

The Plan 9 [24] file system is a one-server system that provides an atomic dump operation for backups. During a dump operation, the in-memory cache is flushed, the file system is frozen, and all blocks modified since the last dump operation are queued on disk for writing to the WORM storage. This system delays access to the file system while the dump is being taken.

The Write Anywhere File Layout (WAFL) [8] is a file system designed for an NFS server appliance that provides a snapshot feature. The first time a page is modified after the snapshot is taken, WAFL writes the page to a new location (thus preserving the old disk copy for the snapshot); it also modifies the disk copy of the directory block that points to this page. The only block that needs to be duplicated eagerly during the snapshot is the root inode block. A snapshot includes the current values of all dirty pages in the cache; if such a page is modified before being written to disk, the proper snapshot value for the page would not be known. To avoid this problem WAFL blocks incoming requests that would modify dirty data that was present in the cache at the moment the snapshot was requested. However, it does not block any other requests.

Frangipani [31] is a distributed file system built on the Petal [12] distributed storage system. Frangipani uses Petal's snapshot features to create a point-in-time copy of the entire distributed file system. This point-in-time copy can then be copied to a tape or WORM device for backup. To maintain consistency of the snapshot, the backup process requests an exclusive lock on the system. This requires stopping the system so that all servers can learn about the snapshot. When it hears about a snapshot, a server flushes its cache to disk and blocks all new file system operations that modify the data. These operations remain blocked until the lock is released. After that point, the backup service takes a Petal snapshot at each server to create the point-in-time copy. This is done copy-on-write without delaying user requests. (Petal doesn't block to take a snapshot but it supports only a single writer.)

Unlike Plan 9, WAFL, and Frangipani, we do not require the system to block users during the snapshot. Furthermore, TimeLine is a distributed system, unlike Plan 9 and WAFL. Frangipani is also distributed but requires a global lock to take a snapshot, which can lead to a substantial delay.

The Elephant file system [26] maintains all versions and uses timestamps to provide a consistent snapshot. A version of a file is defined by the updates between an open and close operation on the file. The file's inode is duplicated when the file is opened and copy-on-write is used to create a premodification version of the file. Concurrent sharing of a file is supported by copying the inode on the first open and appending to the inode-log on the last close.

TimeLine is similar to Elephant in its use of timestamps but it is a distributed system while Elephant is a one-server system. In addition TimeLine makes snapshots on command while Elephant takes them automatically. Taking snapshots on command reduces the size of archive storage. It also arguably provides more useful information, since users can decide what snapshots they want, rather than having to contend with huge amounts of data that is hard to relate to activities that interest them.

Some database systems [22, 27] also provide access to historical information. These systems allow queries based on time information that is either provided explicitly by the user or implicitly by the system. When the time field is overwritten, the old value of the record is retained.

An alternative design for snapshots is to store either an undo or redo log and then materialize the snapshot on demand: with an undo log, the current state would be rolled backward to the desired time line, while with a redo log the initial state would roll forward. Use of such logs was pioneered in database system and some early work on the Postgres Storage Manager [29] even proposed keeping the entire state as a redo log. This approach was subsequently rejected because of the cost of materializing the current system state [30]. These results can also be taken to show that materializing snapshot state is not practical: snapshots must be available for use without a huge delay

9 CONCLUSION

This paper describes TimeLine, an efficient archive service for a distributed storage system. TimeLine allows users to take snapshots on demand. The archive is stored online so that it is easily accessible to users. It enables "time travel" in which a user runs a computation on an earlier system state.

TimeLine allows snapshots to be taken without disruption: user access to the store is not delayed when a snapshot is requested. Yet our scheme provides consistent snapshots; in fact it provides atomicity, which is stronger than consistency. The techniques used in TimeLine will also work in a system that only provides atomicity for individual modifications. Furthermore timestamps can be used to order snapshots in any system that implements logical clocks, and logical clocks are easy and cheap to implement.

TimeLine provides consistent snapshots with minimum impact on system performance: storing snapshot pages occurs in the background. Our scheme is also scalable: snapshots are requested by communicating with a single node and information is propagated to the nodes in the system via gossip.

The main performance goal of our system is that snapshots should not interfere with transactions on the current database and degrade their performance. An additional goal is to provide reasonable accesses to snapshots, i.e., to computations that run in the past. Our experiments show that computations in the past run as fast as in the present when the archive state is co-located with the current system state, and run reasonably well using shared archive storage that is close, e.g., on the same LAN, or connected by a high speed link. The results also show that taking snapshots has negligible impact on the cost of concurrently running computations, regardless of where the archived data is stored.

10 Acknowledgements

The authors gratefully acknowledge the help received from Liuba Shrira, Rodrigo Rodrigues, Sameer Ajmani, our shepherd Jeff Chase, and the referees. This research is supported by NTT and NSF grant ANI-0225660.

References

[1] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. *SIGMOD Record*, 22(2):12–21, Janaury 1994.

[2] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for Distributed Storage Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinchart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1987.

[4] E. Elnozahy, L. Alvisi, Y-M Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[5] S. Ghemawat. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1995.

[6] D. Gifford. Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Parc, March 1983.

[7] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003.

[8] D. Hitz, J. Lau, and M.A. Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994.

[9] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots in the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, El Paso, TX, May 1997.

[10] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 1996.

[13] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the ACM SIGMOD Conference*, Montreal, Canada, June 1996.

[14] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Object in Distributed Systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999.

[15] D. Lomet and B. Salzberg. *Temporal Databases: Theory, Design, and Implementation*, pages 388–417. Addison-Wesley, March 1993.

[16] D. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[17] J. Lundelius. *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms*. PhD thesis, Massachusetts Institute of Technology, 1988.

[18] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, March 1996.

[19] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[20] D. Mills. Network Time Protocol (Version 3) specification, implementation and analysis. Technical Report Network Working Group Report RFC-1305, University of Delaware, Newark, DE, USA, March 1992.

[21] B. Oki and B. Liskov. Viewstamped Replication: A General Primary Copy. In *Proceedings of the 7th Symposium on Principles of Distributed Computing (PODC)*, Toronto, Ontario, Canada, August 1988.

[22] G. Ozsoyoglu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knoweldge and Data Engineering*, 7(4):513–532, August 1995.

[23] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[24] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[25] A. Rowston and P. Drushel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM Internationl Conference on Distributed Systems Platrofms (Middleware)*, Heidelberg, Germany, November 2001.

[26] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Charleston, SC, USA, December 1999.

[27] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Proceedings of the 9th Int'l Conference on Advanced Information Systems Engineering*, Barcelona, Spain, June 1997.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balariishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference*, San Deigo, CA, USA, August 2001.

[29] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very-Large Data Bases*, Brighton, England, UK, September 1987.

[30] M. Stonebraker and J. Hellerstein, editors. *Readings in Database Systems*, chapter 3. Morgan Kaufmann Publishers, Inc., 3rd edition, 1998.

[31] C. Thekkath, T. Mann, and E. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Sanit-Malo, France, October 1997.

Explicit Control in a Batch-Aware Distributed File System

John Bent, Douglas Thain,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny
Computer Sciences Department, University of Wisconsin, Madison

Abstract

We present the design, implementation, and evaluation of the Batch-Aware Distributed File System (BAD-FS), a system designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters distributed across the wide area. BAD-FS consists of two novel components: a storage layer that exposes control of traditionally fixed policies such as caching, consistency, and replication; and a scheduler that exploits this control as necessary for different workloads. By extracting control from the storage layer and placing it within an external scheduler, BAD-FS manages both storage and computation in a coordinated way while gracefully dealing with cache consistency, fault-tolerance, and space management issues in a workload-specific manner. Using both microbenchmarks and real workloads, we demonstrate the performance benefits of explicit control, delivering excellent end-to-end performance across the wide-area.

1 Introduction

Traditional distributed file systems, such as NFS and AFS, are built on the solid foundation of empirical measurement. By studying expected workload patterns [7, 41, 45, 50, 57], researchers and developers have long been able to make appropriate trade-offs in system design, thereby building systems that work well for the workloads of interest. Most previous distributed file systems have been targeted at a particular computing environment, namely a collection of interactively used client machines. However, as past work has demonstrated, different workloads lead to different designs (*e.g.*, FileNet [18] and the Google File System [29]); if assumptions about usage patterns, sharing characteristics, or other aspects of the workload change, one must reexamine the design decisions embedded within distributed file systems.

One area of increasing interest is that of batch workloads. Long popular within the scientific community, batch computing is increasingly common across a broad range of important and often commercially viable application domains, including genomics [3], video production [52], simulation [11], document processing [18], data mining [2], electronic design automation [17], financial services [42], and graphics rendering [36].

Batch workloads minimally present the system with a set of jobs that need to be run and perhaps some ordering among them; in many environments, the approximate run times and I/O requirements are also known in advance. A scheduler uses this information to dispatch jobs so as to maximize throughput.

Batch workloads are typically run in controlled local-area cluster environments. However, organizations that have large workload demands increasingly need ways to share resources across the wide-area, both to lower costs and to increase productivity. One approach to accessing resources across the wide-area is to simply run a local-area batch system across multiple clusters that are spread over the wide-area and to use a distributed file system as a backplane for data access.

Unfortunately, this approach is fraught with difficulty, largely due to the way in which I/O is handled. The primary problem with using a traditional distributed file system is in its approach to *control*: many decisions concerning caching, consistency, and fault tolerance are made *implicitly* within the file system. Although these decisions are reasonable for the workloads for which these file systems were designed, they are ill-suited for a wide-area batch computing system. For example, to minimize data movement across the wide-area, the system must carefully use the cache space of remote clusters; however, caching decisions are buried deep within distributed file systems, thus preventing such control.

To mitigate these problems and enable the utilization of remote clusters for I/O-intensive batch workloads, we introduce the Batch-Aware Distributed File System (BAD-FS). BAD-FS differs from traditional distributed file systems in its approach to control; BAD-FS exposes decisions commonly hidden inside of a distributed file system to an external workload-savvy scheduler. BAD-FS leaves all consistency, caching, and replication decisions to this scheduler, thus enabling *explicit* and workload-specific control of file system behavior.

The main reason to migrate control from the file system to the scheduler is *information* – the scheduler has intimate knowledge of the workload that is running and can exploit that knowledge to improve performance and streamline failure handling. The combination of workload information and explicit control of the file system leads to three distinct benefits over traditional approaches:

• **Enhanced performance.** By carefully managing remote cluster disk caches in a cooperative fashion, and by controlling I/O such that only needed data is transported across the wide-area, BAD-FS minimizes wide-area traffic and improves throughput. Using workload knowledge, BAD-FS further improves performance by using capacity-aware scheduling to avoid thrashing.

• **Improved failure handling.** Using detailed workload information, the scheduler can determine whether to make replicas of data based on the cost of generating it, and not indiscriminately as is typical in many file systems. Data loss is treated uniformly as a performance problem. The scheduler has the ability to regenerate a lost file by rerunning the application that generated it and hence only replicates when the cost of regeneration is high.

• **Simplified implementation.** Detailed workload information allows a simpler implementation. For example, BAD-FS provides a cooperative cache but does not implement a cache consistency protocol. Through exact knowledge of data dependencies, it is the scheduler that ensures proper access ordering among jobs. Previous work has demonstrated the difficulties of building a more general cooperative caching scheme [4, 12].

We demonstrate the benefits of explicit control via our prototype implementation of BAD-FS. Using synthetic workloads, we demonstrate that BAD-FS can reduce wide-area I/O traffic by an order of magnitude, can avoid performance faults through capacity-aware scheduling, and can proactively replicate data to obtain high performance in spite of remote failure. Using real workloads, we demonstrate the practical benefits of our system: I/O-intensive batch workloads can be run upon remote resources both easily and with high performance.

Finally, BAD-FS achieves these ends while maintaining site autonomy and support for unmodified legacy applications. Both of these practical constraints are important for acceptance in wide-area batch computing environments.

The rest of this paper is organized as follows. In Section 2, we describe our assumptions about the expected environment and workload, in Section 3, we discuss the architecture of our system, in Section 4, we present our experimental evaluation, in Section 5, we examine related work, and finally in Section 6, we conclude.

2 Background

In this section, we describe the setting for BAD-FS. We present the expected workloads, basing assumptions on our recent work in batch workload characterization [54]. We describe the computing environment available to users of these workloads and the difficulty they encounter executing such workloads with conventional tools.

2.1 Workloads

As illustrated in Figure 1, these data-intensive workloads are composed of multiple independent vertical sequences of processes that communicate with their ancestors and relatives via private data files. A workload generally consists of a large number of these sequences that are incidentally synchronized at the beginning, but are logically distinct and may correctly execute at a different rate than their siblings. We refer to a vertical slice of the workload as a *pipeline*, a horizontal slice as a *batch*, and the entire set as a *batch-pipelined* workload. Note that "pipeline" is used generically; the processes are *not* connected by UNIX-style pipes but rather communicate through files.

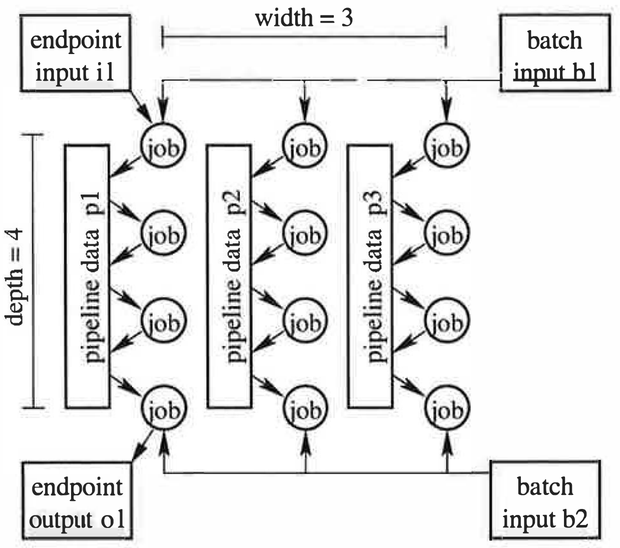


Figure 1: **A Typical Batch-Pipelined Workload.** *A single pipeline represents the logical work that a user wishes to complete, and is comprised of a series of jobs. Users often assemble many such pipelines into a batch to explore variations on input parameters or other input data.*

One of the key differences between a single application and a batch-pipelined workload is file sharing behavior. For example, when many instances of the same pipeline are run, the same executable and potentially many of the same input files are used. We characterize the sharing that occurs in these batch-pipelined workloads by breaking I/O activity into three types (as shown in Figure 1): *endpoint*, the unique input and final output; *pipeline-shared*, shared write-then-read data within a single pipeline; and *batch-shared*, input data shared across multiple pipelines.

2.2 Environment

Although wide-area sharing of untrusted and arbitrary personal computers is a possible platform for batch workloads [53], we believe that a better platform for these types of throughput-intensive workloads is one or more clusters of managed machines, spread across the wide area. We assume that each cluster machine has processing, memory, and local disk space available for remote users, and that each cluster exports its resources via a CPU sharing system. The obvious bottleneck of such a system is the wide-area connection, which must be managed carefully to ensure high performance. For simplicity, we focus most of our efforts on the case of a single cluster being accessed by a remote user. However, in Section 4.8, we present preliminary results from a multi-cluster environment.

We refer to this more organized, less hostile, and well managed collection of clusters as a *c2c (cluster-to-cluster)* system, in contrast to popular peer-to-peer (p2p) systems. Although the p2p environment is appropriate for many uses, there is likely to be a more organized effort to share computing resources within corporations or other organizations. We may assume that c2c environments are more stable, more powerful, and more trustworthy. That said, p2p technologies and designs are likely to be directly applicable to the c2c domain.

We also make the practical and important assumption that each site has local autonomy over its resources. Autonomy has two primary implications on the design of BAD-FS. First, although a workload may be able to use remote resources at a given time, these resources may be arbitrarily revoked. Thus, a system that is built to exploit remote resources must be able to tolerate unexpected resource failures, whether they are due to physical breakdowns, software failures, or deliberate preemptions. Second, autonomy prohibits the deployment of arbitrary software within the remote cluster. In designing BAD-FS, we assume that a remote cluster only provides us with the ability to dispatch a well-defined job as an ordinary, unprivileged user. Mandating that a single distributed file system be used everywhere is not a viable solution.

Finally, we assume that the jobs run on these systems cannot be modified, In our experience, many scientific workloads are the product of years of fine-tuning, and when complete, are viewed as untouchable. Also, ease of use is important; the less work for the user, the better.

2.3 Current Solutions

We now consider a user who wishes to run a batch-pipelined workload in this environment. After the user has developed and debugged the workload on their home system, they are ready to run batches of hundreds or thousands on all available computing resources, using remote batch execution systems such as Condor, LSF, PBS, or Grid Engine. Each pipeline in their workload is expected to use much of the same input data, while varying parameters and other small inputs. The necessary input data begins on the user's *home storage server* (*e.g.*, an FTP server), and the output data, when generated, should eventually be committed to this home server. Conventional batch computing systems present a user with two options for running a workload.

The first option, *remote I/O*, is to simply submit the workload to the remote batch system. With this option, all input and output occur on demand back to the home storage device. Although this approach is simple, the throughput of a data-intensive workload will be drastically reduced by two factors. First, wide-area network bandwidth is not sufficient to handle simultaneous batch reads from many data-intensive pipelines running in parallel. Second, all pipeline output is directed back to the home site, including temporary data that is not needed after the computation completes.

The second option, *pre-staging*, is for the user to manually configure the system to replicate batch data sets in the remote environment. This approach requires the user to have or obtain an account in the remote environment, identify the necessary input data, transfer that data to the remote site, log into the remote system, unpack the data in an appropriate location, configure the workload to recognize the correct directories (possibly using `/tmp` for temporary pipeline data), submit the workload, and manually deal with any failures. The entire process must be repeated whenever more data needs to be processed, new batch systems become available, or existing systems no longer have capacity to offer to the user. As is obvious from the description, this configuration process is labor-intensive and error-prone; additionally, using `/tmp` can be challenging because its availability cannot be guaranteed. Another limitation is that because the user has made these configurations independently of the scheduling system, the scheduling system is not able to correctly checkpoint pipelines within the workload. Still, many users go to these lengths simply to run their workloads.

Traditional distributed file systems would be a better solution but are typically not available due to administrative desire to preserve autonomy across domain boundaries. Even were such systems available, their fixed policies prevent them from being viable for batch-pipelined workloads. Consider, for example, BLAST [3], a commonly used genomic search program, consisting of a single stage pipeline that searches through a large shared dataset for protein string matches. Assume a user were to run BLAST on a compute cluster of 100 nodes equipped with a conventional distributed file system such as AFS or NFS. With cold caches, all 100 nodes will individually (and likely simultaneously) access the home server with the same large demands, resulting in poor performance as the dataset is redundantly transferred over the wide area network. Once the caches are loaded, each node will run at local disk speeds, but only if the dataset can fit in its cache. If it cannot, the node will thrash and generate an enormous amount of repetitive traffic back to the home server. Further, lacking workload information, each node must employ some mechanism to protect the consistency and availability of its cached data.

In contrast, a batch-aware system such as BAD-FS has a global view of the hardware configuration and workflow structure; it can execute such workloads much more efficiently by copying the dataset a single time over the wide

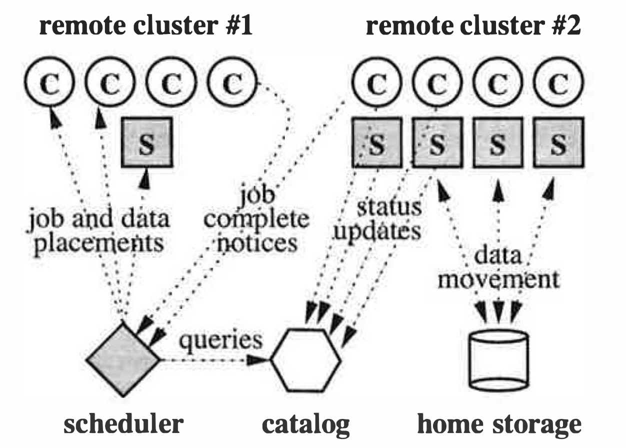


Figure 2: **System Architecture.** *Circles are compute servers, which execute batch jobs. Squares are storage servers, which hold cached inputs and temporary outputs. Both types of servers report to a catalog server, which records the state of the system. The scheduler uses information from the catalog to direct the system by configuring storage devices and submitting batch jobs. The gray shapes are novel elements in our design; the white are standard components found in batch systems.*

area and sharing or duplicating the data at the remote cluster. Further, explicit knowledge of sharing characteristics permits such a system to dispense with the expense and complexity of consistency checks while allowing nodes to continue executing even while disconnected.

3 Architecture

In this section, we present the architecture and implementation of BAD-FS. Recall that the main goal of the design of BAD-FS is to export suf□cient control to a remote scheduler to allow it to deliver improved performance and better fault-handling for I/O-intensive batch workloads run on remote clusters. Figure 2 summarizes the architecture of BAD-FS, with the novel elements shaded gray.

BAD-FS is structured as follows. Two types of server processes manage local resources. A *compute server* exports the ability to transfer and execute an ordinary user program on a remote CPU. A *storage server* exports access to disk and memory resources via remote procedure calls that resemble standard □le system operations. It also permits remote users to allocate space via an abstraction called *volumes*. *Interposition agents* bind unmodi□ed workloads running on compute servers to storage servers. Both types of servers periodically report themselves to a *catalog server*, which summarizes the current state of the system. A *scheduler* periodically examines the state of the catalog, considers the work to be done, and assigns jobs to compute servers and data to storage servers. The scheduler may obtain data, executables, and inputs from any number of external storage sites. For simplicity, we assume the user has all the necessary data stored at a single *home storage server* such as a standard FTP server.

From the perspective of the scheduler, compute and storage servers are logically independent. A specialized device might run only one type of server process: for example, a diskless workstation runs only a compute server, whereas a storage appliance runs only a storage server. However, a typical workstation or cluster node has both computing and disk resources and thus runs both.

BAD-FS may be run in an environment with multiple owners and a high failure rate. In addition to the usual network and system errors, BAD-FS must be prepared to for *eviction* failures in which shared resources may be revoked without warning. The rapid rate of change in such systems creates possibly stale information in the catalog. BAD-FS must also be prepared to discover that the servers it attempts to harness may no longer be available.

BAD-FS makes use of several standard components. Namely, the compute servers are Condor [38] *startd* processes, the storage servers are modi□ed NeST storage appliances [8], the interposition agents are Parrot [55] agents, and the catalog is the Condor *matchmaker*. The servers advertise themselves to the catalog via the ClassAd [43] resource description language.

3.1 Storage Servers

Storage servers are responsible for exporting the raw storage of the remote sites in a manner that allows ef□cient management by remote schedulers. A storage server does not have a □xed policy for managing its space. Rather, it makes several policies accessible to external users who may carve up the available space for caching, buffering, or other tasks as they see □t. Using an abstraction called*volumes*, storage servers allow users to allocate space with a name, a lifetime, and a type that speci□es the policy by which to internally manage the space. The BAD-FS storage server exports two distinct volume types: scratch volumes and cache volumes.

A *scratch volume* is a self-contained read-write □le system, typically used to localize access to temporary data. The scheduler can use scratch volumes for pipeline data passed between jobs and as a buffer for endpoint output. Using scratch volumes, the scheduler minimizes home server traf□c by localizing pipeline I/O and only writing endpoint data when a pipeline successfully completes.

A *cache volume* is a read-only view of a home server, created by specifying the name of the home server and path, a caching policy (*i.e.*, LRU or MRU), and a maximum storage size. Multiple cache volumes can be bound into a *cooperative cache volume* by specifying the name of a catalog server, which the storage servers query to discover their peers. A number of algorithms [16, 20] exist for managing a cooperative cache, but it is not our intent to explore the range of these algorithms here. Rather, we describe a reasonable algorithm for this system and explain how it is used by the scheduler.

The cooperative cache is built using a distributed hash table [31, 37]. The keys in the table are block addresses, and the values specify which server is primarily responsible for that block. To avoid wide-area traffic, only the primary server will fetch a block from the home server and the other servers will create secondary copies from the primary. When space is needed, secondary data is evicted before primary. To approximate locality, our initial implementation only forms cooperative caches between peers in the same subnetwork. Although our initial analysis suggests that this is sufficient, in the future we plan on investigating other more complicated grouping algorithms.

Failures within the cooperative cache, including partitions, are easily managed but may cause slowdown. Should a cooperative cache be internally partitioned, the primary blocks that were assigned to the now missing peers will be reassigned. As long as the home server is accessible, partitioned cooperative caches will be able to refetch any lost data and continue without any noticeable disturbance to running jobs.

This approach to cooperative caching has two important differences from previous work. First, because data dependencies are completely specified by the scheduler, we do not need to implement a cache consistency scheme. Once read, all data are considered current until the scheduler invalidates the volume. This design decision greatly simplifies our implementation; previous work has demonstrated the many difficulties of building a more general cooperative caching scheme [4, 12]. Second, unlike previous cooperative caching schemes that manage cluster memory [16, 20], our cooperative cache stores data on local *disks*. Although managing memory caches cooperatively could also be advantageous, the most important optimization to make in our environment is to avoid data movement across the wide-area; managing remote disk caches is the simplest and most effective way to do so.

3.1.1 Local vs. Global Control. Note that volumes export only a certain degree of control to the scheduler. Namely, by creating and deleting volumes, the scheduler controls which data sets reside in the remote cluster. However, the storage servers retain control over per-block decisions. Two such important decisions made locally by the storage servers are the assignment of primary blocks in the cooperative cache and cache victim selection. Of course, if the scheduler is careful in space allocation, the cache will only victimize blocks that are no longer needed. In general, we have found this separation of global and local control to be suitable for our workloads. Although more work needs to be done to precisely identify the balance point, it is clear that a trade-off is better than either extreme. Complete local control, the current approach, suffers because the policies embedded within distributed file systems are inappropriate for batch workloads. The other extreme, complete global control, in which the scheduler makes decisions for each block of data, would require exorbitant complexity in the scheduler and would incur excessive network traffic to exert this fine-grained control.

3.2 Interposition Agents

In order to permit ordinary workloads to make use of storage servers, an *interposition agent* [33] transforms POSIX I/O operations into storage server calls. The agent's mapping from logical path names to physical storage volumes is provided by the scheduler at runtime. Together, the agent and the volume abstraction can hide a large number of errors from the job and the end user. For example, if a volume no longer exists, whether due to accidental failure or deliberate preemption, a storage server returns a unique *volume lost* error to the agent. Upon discovering this, the agent forcibly terminates the job, indicating that it could not run correctly in the given environment. This gives the scheduler clear indication of failures and allows it to take transparent recovery actions.

3.3 The Scheduler

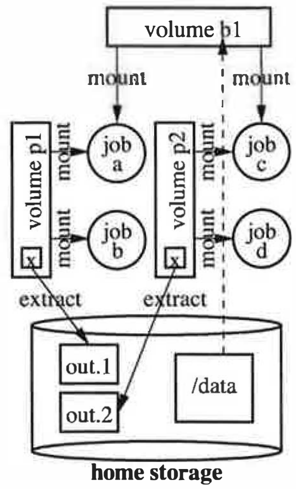
The BAD-FS scheduler directs the execution of a workload on compute and storage servers by combining a static workload description with dynamic knowledge of the system state. Specifically, the scheduler minimizes traffic across the wide-area by differentiating I/O types and treating each appropriately, carefully managing remote storage to avoid thrashing and replicating output data proactively if that data is expensive to regenerate.

3.3.1 Workflow language. Shown in Figure 3 is an example of the declarative workflow language that describes a batch-pipelined workload and shows how the scheduler converts this description into an execution plan. The keyword `job` names a job and binds it to a description file, which specifies the information needed to execute that job. `Parent` indicates an ordering between two jobs. The `volume` keyword names the data sources required by the workload. For example, volume `b1` comes from an FTP server, while volumes `p1` and `p2` are empty scratch volumes. Volume sizes are provided to allow the scheduler to allocate space appropriately. The `mount` keyword binds a volume into a job's namespace. For example, jobs `a` and `c` access volume `b1` as `/mydata`, while jobs `a` and `b` share volume `p1` via the path `/tmp`. The `extract` command indicates which files of interest must be committed to the home server. In this case, each pipeline produces a file `x` that must be retrieved and uniquely renamed.

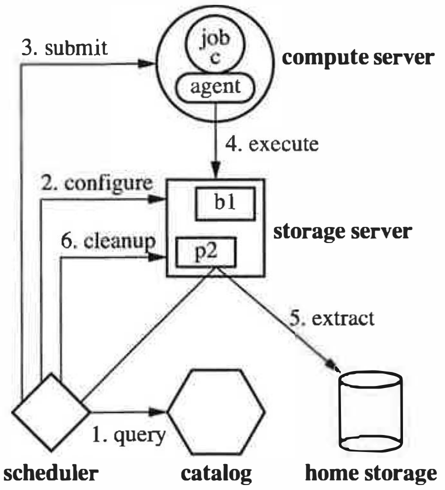
To many readers accustomed to working in an interactive environment, this language may seem like an unusual burden. We point out that a user intending to execute batch-pipelined workloads must be exceptionally organized. Batch users already provide this information, but it is scattered across shell scripts, make files, and batch submission files. In addition to imparting needed information to the BAD-FS scheduler, this workflow language actually reduces user burden by collecting all of this dispersed information into a coherent whole.

```
job      a    a.condor
job      b    b.condor
job      c    c.condor
job      d    d.condor
parent   a    child    b
parent   c    child    d
volume   b1   ftp://home/data 1 GB
volume   p1   scratch 50 MB
volume   p2   scratch 50 MB
mount    b1   a    /mydata
mount    b1   c    /mydata
mount    p1   a    /tmp
mount    p1   b    /tmp
mount    p2   c    /tmp
mount    p2   d    /tmp
extract  p1   x    ftp://home/out.1
extract  p2   x    ftp://home/out.2
```

(A)



(B)



(C)

Figure 3: **Workflow and Scheduler Examples.** (A) *A simple workflow script. A directed graph of jobs is constructed using* job *and* parent, *and the file system namespace presented to jobs is configured with* volume *and* mount. *The* extract *keyword indicates which files must be commited to the home storage server after pipeline completion.* (B) *A graphical representation of this workflow.* (C) *The scheduler's plan for job c. (1) The scheduler queries the catalog for the current system state and decides where to place job c and its data. (2) The scheduler creates volumes b1 and p2 on a storage server. (3) Job c is dispatched to the compute server. (4) Job c executes, accessing its volumes via the agent. (5) After jobs c and d complete, the scheduler extracts x from p2. (6) The scheduler frees volumes b1 and p2.*

3.3.2 I/O Scoping. Unlike most file systems, BAD-FS is aware of the flow of its data. From the workflow language, the scheduler knows where data originates and where it will be needed. This allows it to create a customized environment for each job and minimize traffic to the home server. We refer to this as *I/O scoping*.

I/O scoping minimizes traffic in two ways. First, cooperative cache volumes are used to hold read-only batch data such as `b1` in Figure 3. Such volumes may be reused without modification by a large number of jobs. Second, scratch volumes, such as `p2` in Figure 3, are used to localize pipeline data. As a job executes, it accesses only those volumes that were explicitly created for it; the home server is accessed only once for batch data and not at all for pipeline.

3.3.3 Consistency management. With the workload information expressed in the workflow language, the scheduler neatly addresses the issue of consistency management. All of the required dependencies between jobs and data are specified directly. Since the scheduler only runs jobs so as to meet these constraints, there is no need to implement a cache consistency protocol among the BAD-FS storage servers.

The user may make mistakes in the workflow description that can affect both cache consistency and correct failure recovery. However, through an understanding of the expected workload behavior as specified by the user, the scheduler can easily detect these mistakes and warn the user that the results of the workload may have been compromised. We have not yet implemented these detection features, but the architecture readily admits them.

3.3.4 Capacity-Aware Scheduling. The scheduler is responsible for throttling a running workload to avoid performance faults and maximize throughput. By carefully allocating volumes, the scheduler avoids overflowing storage or thrashing caches. Although disk capacity is rapidly increasing, the size of data sets is also growing and space management remains important [4, 6, 23]. The scheduler manages space by retrieving a list of available storage from the catalog server and selecting the ready job with the least unfulfilled storage needs, whether pipe or batch. If the scheduler is able to allocate all of that job's volumes, then it allocates and configures these volumes and schedules the job. If there are no jobs to execute or not enough available space, then the scheduler waits for a job to complete, more resources to arrive, or for a failure to occur. Note that due to a lack of complete global control, the scheduler may need to slightly overprovision when the needed volume size approaches the storage capacity.

In other scheduling domains, selecting the smallest job first can result in starvation. In this domain, however, starvation is avoided because a workflow is a static entity executed by one scheduler. Although smaller jobs will run first, all jobs will eventually be run.

3.3.5 Failure Handling. Finally, the scheduler makes BAD-FS robust to failures by handling failures of jobs, storage servers, the catalog, and itself. One aspect of batch workloads that we leverage is job idempotency; a job can simply be rerun in order to regenerate its output.

The scheduler keeps a log of allocations in persistent storage, and uses a transactional interface to the compute and storage servers. If the scheduler fails, then allocated volumes and running jobs will continue to operate unaided. If the scheduler recovers, it simply re-reads the log to discover what resources have been allocated and resumes normal operations. Recording allocations persistently allows them to be either re-discovered or released in a timely manner. If the log is irretrievably lost, then the workflow must be resumed from the beginning; previously acquired leases will eventually expire.

In contrast, the catalog server uses soft state. Since the catalog is only used to discover new resources, there is no need to recover old state from a crash. When the catalog is unavailable, the scheduler will continue to operate on known resources, but will not discover new ones. When the catalog server recovers, it rebuilds its knowledge as compute and storage servers send periodic updates.

The scheduler waits for passive indications of failure in compute and storage servers and then conducts active probes to verify. For example, if a job exits abnormally with an error indicating a failure detected by the interposition agent, then the scheduler suspects that the storage servers housing one or more of the volumes assigned to the job are faulty. The scheduler then probes the servers. If all volumes are healthy, it assumes the job encountered transient communication problems and simply reruns it. However, if the volumes have failed or are unreachable for some period of time, they are assumed lost.

The failure of a volume affects the jobs that use it. As a design simplification, the scheduler considers a partial volume failure to be a failure of the entire volume; in the future we plan to investigate the trade-offs involved in the choice of failure granularities. Running jobs that rely on a failed volume must be stopped. In addition, failures can cascade; completed processes that *wrote* to a volume must be rolled back and re-run. In order to avoid these expensive restarts of a pipeline, the scheduler may checkpoint scratch volumes as pipeline stages complete.

Of course, determining an optimal checkpoint interval is an old problem [27]. The solution depends upon the likelihood of failure, the value of a checkpoint, and the cost to create it. Unlike most systems, BAD-FS can solve this problem automatically, because the scheduler is in a unique position to measure the controlling variables. The scheduler performs a simple cost-benefit analysis at runtime to determine if a checkpoint is worthwhile.

The algorithm works as follows. The scheduler tracks the average time to replicate a scratch volume. This cost is initially assumed to be zero in order to trigger at least one replication and measurement. To determine the benefit of replication, the scheduler tracks the number of job and storage failures and computes the mean-time-to-failure across all devices in the system. The benefit of replicating a volume is the sum of the run times of those jobs completed so far in the applicable pipeline multiplied by the probability of failure. If the benefit exceeds the cost, then the scheduler replicates the volume on another storage server as insurance against failure. If the original fails, the scheduler restarts the pipeline using the saved copy.

Due to its robust failure semantics, the scheduler need not handle network partitions any differently than other failures. When partitions are formed between the scheduler and compute servers, the scheduler may choose to reschedule any jobs that were running on the other side of the partition. In such a situation, it is possible that the partition could be resolved, at which point the scheduler will find that multiple servers are executing the same jobs. Note that this will not introduce errors because each job writes to distinct scratch volumes. The scheduler may choose one output to extract and then discard the other.

3.4 Practical Issues

One of the primary obstacles to deploying a new distributed system is the need for a friendly administrator. Whether deploying an operating system, a file system, or a batch system, the vast majority of such software requires a privileged user to install and oversee the software. Such requirements make many forms of distributed computing a practical impossibility; the larger and more powerful the facility, the more difficult it is for an ordinary user to obtain administrative privileges. To this end, BAD-FS is packaged as a *virtual batch system* that can be deployed over an existing batch system without special privileges. This technique is patterned after the "glide-in job" described by Frey *et al.* [26] and is similar in spirit to recursive virtual machines [22].

To run BAD-FS, an ordinary user need only to be able to submit jobs into an existing batch system. BAD-FS bootstraps itself on these systems, relying on the basic ability to queue and run a self-extracting executable program containing the storage and compute servers and the interposition agent. Once deployed, the servers report to a catalog server, and the scheduler may then harness their resources. Note that the scheduling of the virtual batch jobs is at the discretion of the host system; these jobs may be interleaved in time and space with jobs submitted by other users. We have used this technique to deploy BAD-FS over several existing Condor and PBS batch systems.

Another practical issue is security. BAD-FS currently uses the Grid Security Infrastructure (GSI) [24], a public key system that delegates authority to remote processes through the use of time-limited proxy certificates. To bootstrap the system, the submitting user must enter a password to unlock the private key at his/her home node and generate a proxy certificate with a user-settable timeout. The proxy certificate is delegated to the remote system and used by the storage servers to authenticate back to the home storage server. This requires that users trust the host system not to steal their secrets, which is reasonable in a c2c environment.

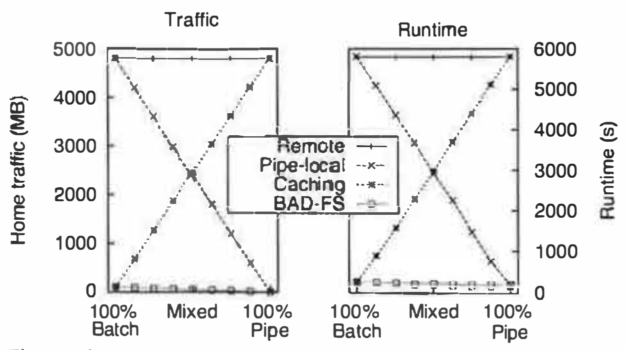


Figure 4: **I/O Scoping: Traffic Reduction and Run Times.** *These graphs show the total amount of network traffic generated by and runtimes for a number of different workloads with different optimizations enabled. For this experiment, we run 48 synthetic pipelines of depth 4, each of which generates a total of 100 MB I/O. Across the x-axis we vary the relative amounts of batch I/O and pipeline I/O. For example, at 100% Batch, the workload generates 100 MB of batch I/O and no pipeline. As is common in these types of workloads, the amount of endpoint I/O is small (1 KB). The leftmost graph shows the total amount of home server traffic; the right shows total runtimes when the home server is accessed over an emulated wide-area network (set at 1 MB/s).*

4 Experimental Evaluation

In this section, we present an experimental evaluation of BAD-FS under a variety of workloads. We first present our methodology, and then focus on I/O scoping, capacity-aware scheduling, and failure handling, using synthetic workloads to understand system behavior. Second, we present our experience with running real workloads on our system in a controlled environment. Finally, we discuss our initial experience using BAD-FS to run real workloads across multiple clusters in the wild.

4.1 Methodology

In the initial experiments in this section, we build an environment similar to that described in Section 2. We assume the user's input data is stored on a home server; once all pipelines have run and all output data is safely stored back at the home server, the workload is considered complete.

We assume that the workload is run on a remote cluster of machines, accessible from the user's home server via a wide-area link. To emulate this scenario, we limit the bandwidth to the home server to 1 MB/s via a simple network delay engine similar to DummyNet [44]. Thus, all I/O between the remotely run jobs and the home server must traverse this slow link. The cluster itself is taken from a dedicated compute pool of Condor nodes at the University of Wisconsin, connected via a 100 Mbit/s Ethernet switch. Each node has two Pentium-3 processors, 1 GB of physical memory and a 9 GB IBM SCSI drive, of which only a 1 GB partition is made available to Condor jobs. Of these 1 GB partitions, typically only about half is available at any one time as the rest awaits lazy garbage collection.

To explore the performance of BAD-FS under a range of workload scenarios, we utilize a parameterized synthetic batch-pipelined workload. The synthetic workload can be configured to perform varying amounts of endpoint, batch, and pipeline I/O, compute for different lengths of time, and can exhibit different amounts of both batch and pipeline parallelism. As each experiment requires different parameters, we leave those descriptions for the individual figure captions. However, given our previous results in workload analysis [54], we focus on *batch-intensive* workloads, which exhibit a high degree of batch sharing but little pipeline or endpoint I/O, and *pipe-intensive*, which perform large amounts of pipeline I/O but generate little batch or endpoint I/O.

4.2 I/O Scoping

The results of the first experiment, as shown in Figure 4, demonstrate how BAD-FS uses I/O scoping to minimize traffic across the wide area by localizing pipeline I/O in scratch volumes and reusing batch data in cooperative cache volumes. Although these optimizations are straightforward, their ability to increase throughput is significant.

In this experiment, we repeatedly run the same synthetic workload but vary the relative amount of batch and pipeline I/O. We compare a number of different system configurations. In the *remote* configuration, all I/O is sent to the home node. Against this baseline, we compare the *pipeline localization* and *caching* optimizations. Finally, both optimizations are combined in the BAD-FS configuration. Note that in these experiments, we assume copious cache space and a controlled environment; neither capacity-aware scheduling nor failure recovery is needed.

The left-hand graph shows the total I/O that is transferred over the wide-area network. Not surprisingly, the cooperative cache greatly reduces batch traffic to the home node by ensuring that all but the first reference to a batch data set is retrieved from the cache. We can also see that the pipeline localization optimizations work as expected, removing pipeline I/O entirely from the home server. Finally, we see that neither optimization in isolation is sufficient; only the BAD-FS configuration that combines both is able to minimize network traffic throughout the entire workload range. The right-hand graph in Figure 4 shows the runtimes of the workloads on our emulated remote cluster. From this graph, we can see the direct impact that wide-area traffic has on runtime.

4.3 Capacity-Aware Scheduling

Next, we examine the benefits of explicit storage management. The previous experiments were run in an environment where storage was not used to near capacity. With the increasing size of batch data sets and storage sharing by jobs and users, the scheduler must carefully manage remote space so as to avoid wide-area thrashing.

For these experiments, we compare the capacity-aware BAD-FS scheduler to two simple variants: a *depth-first* scheduler and a *breadth-first* scheduler. These algorithms are not aware of the data needs of the workload and

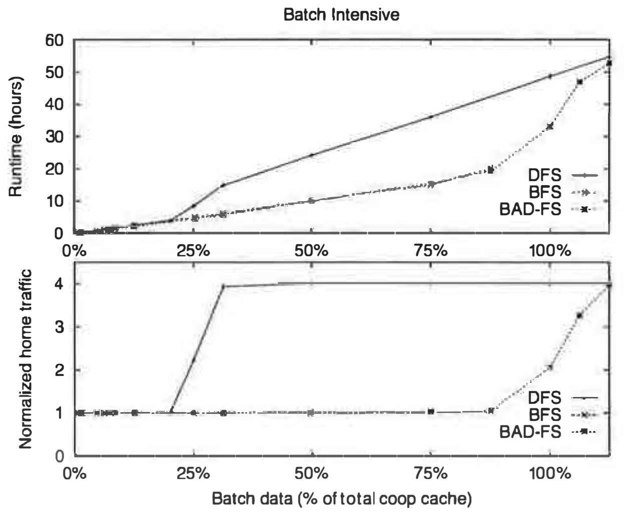


Figure 5: **Batch-intensive Explicit Storage Management.** *These graphs show the benefits of explicit storage management under a batch-intensive workload. The workload consists of 32 4-stage pipelines; within each stage, each process streams through a shared batch file (i.e., there are 4 batch files total). Batch file size is varied as a percentage of the total amount of cooperative cache space available across the 16 nodes in the experiment. All other I/O amounts are negligible. Each of 16 nodes has local storage which is used as a portion of the cache. The total cache size available is set to 8 GB (100% on the x-axis), which reflects our observations of available storage in the UW Condor pool.*

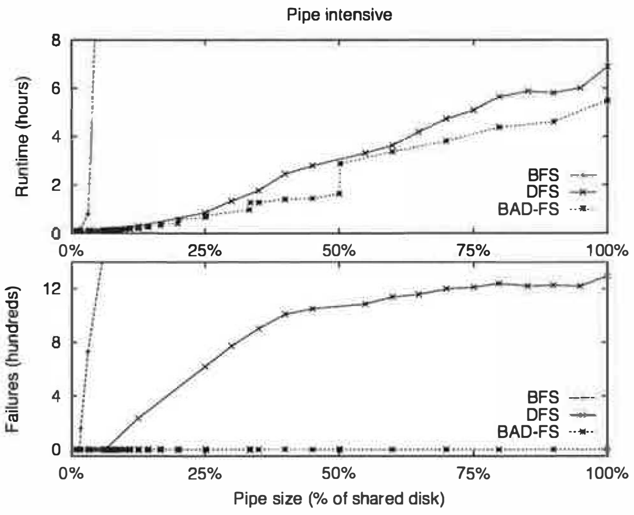


Figure 6: **Pipe-intensive Explicit Storage Management.** *These graphs depict the benefits of explicit storage management under a pipe-intensive workload. The workload consists of 32 4-stage pipelines, and pipe data size is varied as a percent of total storage available. All other I/O amounts are negligible. There are 16 compute servers and 1 storage server in this experiment (representing a set of diskless clients and a single server). The storage space at the server is constrained to 512 MB.*

base decisions solely on the job structure of its workflow. Depth-first simply assigns a single pipeline to each available CPU and runs all jobs in the pipeline to completion before starting another. Conversely, breadth-first attempts to execute all jobs in a batch to completion before descending to the next horizontal batch slice.

Each is correct for certain types of workloads, but can lead to poor storage allocations in others. For example, depth-first scheduling of a batch-intensive workload is more likely to cause thrashing because it attempts to simultaneously cache all of the batch datasets. Similarly, breadth-first scheduling of a pipe-intensive workload is more likely to over-allocate storage because it creates allocations for all pipelines before completing any.

4.4 Batch-intensive Capacity-Aware Scheduling

Figure 5 illustrates the importance of capacity-aware scheduling through measurements of batch-intensive workloads scheduled using various algorithms. Each workload is of depth four and thus has four large batch data sets, each of which takes up some sizable fraction of the available cooperative cache in the remote cluster (as varied along the x-axis). The upper graph shows the runtime and the lower presents the amount of wide-area traffic generated, normalized to the size of the batch data.

We can make a number of observations from these graphs. First, the similarity between the graphs validates that the wide-area network link is the bottleneck resource. Second, as expected, the different policies achieve similar results as long as the entirety of all four batch data sets fits within the caches (*i.e.*, up to 25%). As the size of the batch data approaches the total capacity of the cooperative cache, the runtime and wide-area traffic increase for depth-first scheduling. As the total batch data no longer fits in cache, depth-first scheduling must refetch batch data for each pipeline. In this case, this results in three extra fetches because with 64 pipelines and 16 compute servers, each server executes four pipelines.

Third, note that the runtime actually begins to increase slightly before 25%. The reason for this inefficiency is the lack of complete global control allowed through the current volume interface. In this case, the local cooperative cache hash function is not perfectly distributing data across its peers; when the cache nears full utilization, this skew overloads some nodes and results in extra traffic to the home server. Because we believe that this trade-off between local and global control is correct, the implication here is that the scheduler must be aware not only of the overall utilization of the cooperative cache, but also of the utilization of each peer.

Finally, breadth-first and BAD-FS scheduling are able to retain linear performance in this regime because they ensure that the total amount of batch data accessed at any one time does not exceed the capacity of the cooperative cache. However, once each individual batch dataset exceeds the capacity of the cooperative cache, the performance of breadth-first and BAD-FS scheduling converges with that of depth-first. Note that the same inefficiency that caused depth-first to deviate slightly before 25% causes this to happen slightly before 100%.

4.5 Pipe-intensive Capacity-Aware Scheduling

In our next set of cache management experiments, we focus on a pipeline-intensive workload instead of a batch-

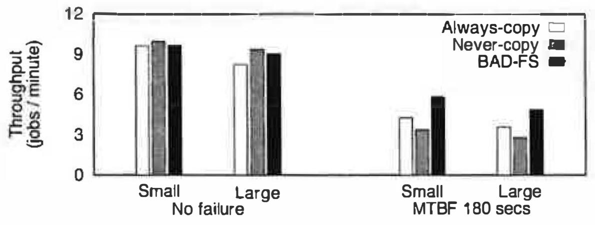


Figure 7: **Failure Handling.** *This graph shows the behavior of the cost-benefit strategy under different failure scenarios. Shown are two different workloads of width 64, depth 3 and one minute of CPU time; one performs a small amount of pipeline I/O, the other a large amount. Each is run both during periods of high and low rates of failure. Failures were induced using an artificial failure generator which formatted disks at random with a mean time between failures of 180 seconds, corresponding to the total runtime of a single pipe.*

intensive one. In this case, we expect the capacity-aware approach to follow the depth-first strategy more closely. Results are presented in Figure 6.

In the lower graph, we plot the number of failed jobs that each strategy induces. Job failure arise in this workload when there is a shortage of space for pipeline output; in such a scenario, a job that runs out of space for pipeline data aborts and must be rerun at some later time. Hence, the number of job failures due to lack of space is a good indicator of the scheduler's success in scheduling pipeline-intensive jobs under space constraints.

From the graph, we can observe that breadth-first scheduling is unable to prevent thrashing. In contrast, the capacity-aware BAD-FS scheduler does not exceed the available space for pipelines and thus never observes an aborted job. This careful allocation results in a drastically reduced runtime which is shown in the upper graph.

The stair-step pattern in the runtime of BAD-FS results from this careful allocation. When the size of the data in each pipeline is between 25% and 33% of the total storage, BAD-FS schedules workload jobs on only 3 of the 16 available CPUs; between 33% and 50% on just two; and as the data exceeds 50%, BAD-FS allocates only a single CPU at a time. Notice that BAD-FS achieves runtimes comparable or better than that of depth-first scheduling without any wasted resource consumption.

4.6 Failure Handling

We now show the behavior of BAD-FS under varying failure conditions. Recall that unlike traditional distributed systems, the BAD-FS scheduler knows exactly how to re-create a lost output file; therefore, whether to make a replica of a file on the remote cluster should depend on the cost of generating the data versus the cost of replicating it. This choice varies with the workload and the system conditions. Figure 7 shows how the BAD-FS cost-benefit analysis adapts to a variety of workloads and conditions. We compare to two naive algorithms: *always-copy*, which replicates a pipeline volume after each of its stages completes and *never-copy*, which does not replicate at all.

We draw several conclusions from this graph. In an environment without failure, replication leads to excessive overhead that increases with the amount of data. In this case BAD-FS outperforms always-copy but does not quite match never-copy because of the initial replication it needs to seed its analysis. In an environment with frequent failure, it is not surprising that BAD-FS outperforms never-copy. Less intuitively, BAD-FS also outperforms always-copy. In this case, given the particulars of the workload and the failure rate, replicating is only worthwhile after the second stage; BAD-FS correctly avoids replicating after the first stage while always-copy naively replicates after all stages.

4.7 Workload Experience

We conclude with demonstrations of the system running real workloads. In the first demonstration as presented in Figure 8, we compare the runtime performance of BAD-FS to other methods of utilizing local storage resources. In the *remote* configuration, local storage is not utilized at all and all I/O is executed directly at the home node. *Standalone* emulates AFS by caching data at the execute nodes but without any cooperative caching among their storage servers. The leftmost graph shows results for remote workload execution in which the bandwidth to the home server was constrained at 1 MB/s; the rightmost shows local workload execution in which the home server was situated within the same local area network as the execute nodes.

From these graphs, we can draw several conclusions. First, BAD-FS equals or exceeds the performance of remote I/O or standalone caching for all of the workloads in all of the configurations. These workloads, which are discussed in great detail in our earlier profiling work [54], all have large degrees of either batch or pipeline data sharing. Note that workloads whose I/O consists entirely of endpoint data would gain no benefit from our system.

Second, the benefit of caching, either cooperatively or in standalone mode, is greater for batch-intensive workloads, such as BLAST, than it is for more pipe-intensive ones such as HF. In these pipe-intensive workloads, the important optimization is pipeline localization, which is performed by both BAD-FS and standalone.

Third, cooperative caching in BAD-FS can outperform standalone both during cold and warm phases of execution. If the entire batch data set fits on each storage server, then cooperative caching is only an improvement while the data is being initially paged in. However, should the data exceed the capacity of any of the caches, then cooperative caching, unlike standalone, is able to aggregate the cache space and fit the working set.

This benefit of cooperative caching with warm caches is illustrated in the BLAST measurements in the graph on the left of Figure 8. Logfile analysis showed that

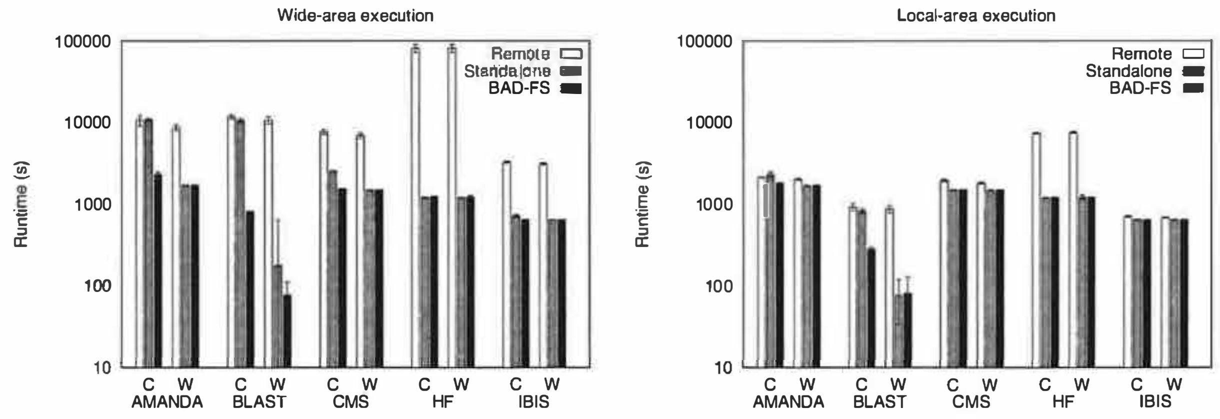


Figure 8: **Workload Experience.** *These graphs show runtime measurements of real workloads. For each workload, we submit 64 pipelines into a dedicated Condor pool of 16 CPUs. This Condor pool accesses local storage resources in one of three configurations:* remote *in which all I/O is redirected back to the home node;* standalone, *which emulates AFS-like caching to the home server; and* BAD-FS. *For each measurement, we present average runtime for the first jobs to run on each storage server when the storage cache is* cold *(C) and for the subsequent jobs which run when the cache is* warm *(W). The graph on the left shows runtimes when the workload is executed on a cluster separated from the home node by an emulated wide-area link (again set to 1 MB/s). On the right the home node is located within the same local area network. Note that the y-axis is shown in log scale to accentuate points of interest. Detailed information about these workloads can be found in our profiling study [54].*

two of the storage servers had slightly less cache space (≈500 MB) than was needed for the total BLAST batch data (≈600 MB). As subsequent jobs accessed these servers, they were forced to refetch data. Refetching it from the wide-area home server in the standalone case was much more expensive than refetching from the cooperative cache as in BAD-FS. With a local-area home server this performance advantage disappears. The different behavior of these two servers also explains the increased variability shown in these measurements.

Fourth, the penalty for performing remote I/O to the home node is less severe but still significant when the home node is in the same local-area network as the execute cluster. This result illustrates that BAD-FS can improve performance even when the bandwidth to the home server is not obviously a limiting resource.

Finally, comparing across graphs we make the further observation that BAD-FS performance is almost independent of the connection to the home server when caches are cold and becomes independent once they are warm. Using I/O scoping, BAD-FS is able to achieve local performance in remote environments.

4.8 In the Wild

Thus far, our evaluations have been conducted in controlled environments. We conclude our experimental presentation with a demonstration that BAD-FS is capable of operating in an uncontrolled, real world environment.

We created a wide-area BAD-FS system out of two existing batch systems. At the University of Wisconsin (UW), a large Condor system of over one thousand CPUs, including workstations, clusters, and classroom machines, is shared among a large number of users. At the University of New Mexico (UNM), a PBS system manages a cluster of over 200 dedicated machines.

We established a personal scheduler, catalog, and home storage server for our use at Wisconsin and then submitted a large number of BAD-FS bootstrap jobs to both batch systems without installing any special software at either of the locations. We then directed the scheduler to execute a large workload consisting of 2500 CMS pipelines using whatever resources became available.

Figure 9 is a timeline of the execution of this workload. As expected, the number of CPUs available to us varied widely, due to competition with other users, the availability of idle workstations (at UW), and the vagaries of each batch scheduler. UNM consistently provided twenty CPUs, later jumping to forty after nine hours. Two spikes in the available CPUs between 4 and 6 hours are due to the crash and recovery of the catalog server; this resulted in a loss of monitoring data, but not of running jobs.

The benefits of cooperative caching are underscored in such a dynamic environment. In the bottom graph, the cumulative read traffic from the home node is shown to have several hills and plateaus. The hills correspond to large spikes in the number of available CPUs.

Whenever CPUs from a new subnet begin executing, they fetch the batch data from the home node. However, smaller hills in the number of available CPUs do not have an effect on the amount of home read traffic because a new server entering an already established cooperative cache is able to fetch most of the batch data from its peers.

Finally, Figure 9 illustrates that both the design and implementation of BAD-FS are suitable for running I/O intensive, batch-pipelined workloads across multiple, uncontrolled real world clusters. Through failures and disconnections, BAD-FS continues making steady progress, removing the burden from the user of scheduling, monitoring, and resubmitting jobs.

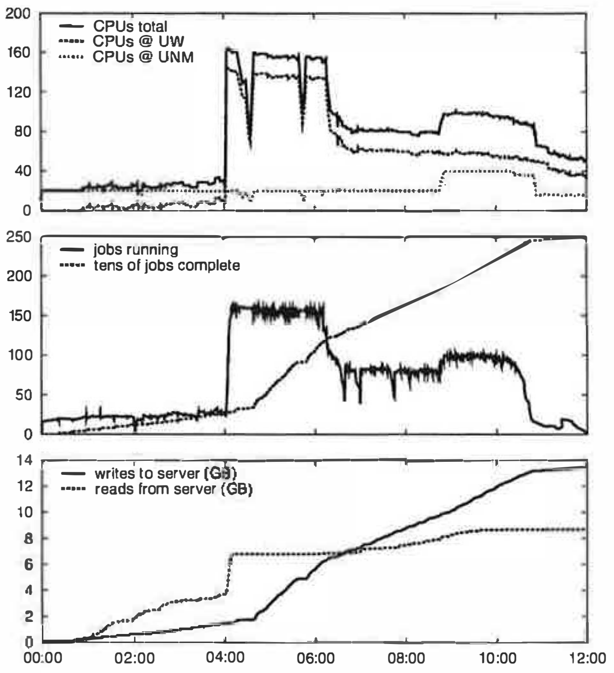


Figure 9: **In the Wild** *These three graphs present a timeline of the behavior of a large CMS workload run using BAD-FS. The workload consisted of 2500 CMS pipelines and was run wherever resources could be scavenged from a collection of CPUs at the University of New Mexico running PBS and from CPUs at the University of Wisconsin running Condor. The topmost timeline presents the total number of CPUs, the middle shows number of jobs running and cumulative jobs completed, and the bottom shows the cumulative traffic incurred at the home server.*

5 Related Work

In designing BAD-FS, we drew on related work from a number of distinct areas. Workflow management has historically been the concern of high-level business management problems involving multiple authorities and computer systems in large organizations, such as approval of loans by a bank or customer service actions by a phone company [28]. Our scheduler works at a lower semantic level than such systems; however, it does borrow several lessons from them, such as the integration of procedural and data elements [47]. The automatic management of dependencies for both performance and fault tolerance is found in a variety of tools [10].

Many other systems have also managed dependencies among jobs. A basic example is found with the UNIX tool `make`. More elaborate dependency tracking has been explored in Vahdat and Anderson's work on transparent result caching [56]; in that work, the authors build a tool that tracks process lineage and file dependency automatically. Our workflow description is a static encoding of such knowledge.

The manner in which the scheduler constructs private namespaces for running workloads is reminiscent of database views [32]. However, a private namespace is simpler to construct and maintain; views, in contrast, present systems with many implementation challenges, particularly when handling updates to base tables and their propagation into extant materialized views.

BAD-FS could be further improved through the prefetching of batch datasets. Other work [13] has noted the difficulty in correctly predicting future access patterns. In BAD-FS, however, these are explicitly supplied by the user via the declarative workflow description.

There has been much recent work in peer-to-peer storage systems [1, 4, 15, 35, 39, 46, 48]. Although each of these systems provides interesting solutions to the problem domain for which they are intended, each falls short when applied to the context of batch workloads, for the same reasons that distributed file systems are not a good match. However, many of the overlays developed for these environments may be useful for communication between clusters, something we plan to investigate in future work. Similar to p2p is work within grid computing [25], which uses many of the same techniques but is designed, as is BAD-FS, for c2c environments. One such example is Cluster-on-Demand [14] which offers sophisticated resource clustering techniques that could be used by BAD-FS to form cooperative cache groupings.

Extensible systems also share our approach of allowing the application more control [9, 19, 51]. Although recent work has recently revisited this approach [5], extensible systems have not been commercially successful because the need for specialized policies is not so great. We believe this need is greater for batch workloads running on systems designed for interactive use.

Some research in mobile computing bears similarity as well. Flinn *et al.* discuss the process of data staging on untrusted surrogates [21]. In many ways, such a surrogate is similar to the BAD-FS storage server; the major difference is that the surrogate is primarily concerned with trust, whereas our servers are primarily concerned with exposing control. Both Zap [40] and VMWare [49] allow for the checkpointing and migration of either processes or operating systems. We create a remote virtual environment, but at the much higher level of a batch system. Systems with secure interposition such as Janus [30] complement BAD-FS as they should make resource owners more willing to donate their resources into shared pools.

Finally, BAD-FS is similar to other distributed file systems. The Google File System [29] was also motivated by workloads that deviate from earlier file system assumptions. An additional similarity is a simplified consistency implementation; however, GFS must relax consistency semantics to enable this, while BAD-FS does so through explicit control. Earlier work on Coda, and AFS before it, is also applicable [34]. These systems use caching for availability, so as to allow disconnected operation. In BAD-FS, storage servers enact a similar role.

6 Conclusions

"He's a big bad wolf in your neighborhood;
not bad meaning bad but bad meaning good."
Run DMC, from 'Peter Piper'

Allowing external control has long been recognized as a powerful technique to improve many aspects of system performance. By moving control to the external user of a system, that system allows the user to dictate the policy that is most appropriate to the individual nature of their work. Systems lacking mechanisms for external control can only speculate. However, many systems have proven to be adept at speculation and work well for the majority of their workloads. In this paper we have argued that the distinct nature of batch workloads is not well matched by the design of traditional distributed file systems and the need therefore for external control is greater.

We have described BAD-FS, a distributed file system that exposes internal control decisions to an external scheduler. Using detailed knowledge of workload characteristics, the scheduler carefully manages remote resources and facilitates the execution of I/O intensive batch jobs on both wide-area and local-area clusters. Through synthetic and real workload measurements in both controlled and uncontrolled environments, we have demonstrated the ability of BAD-FS to use workload specific knowledge to improve throughput by selecting appropriate storage policies in regards to I/O scoping, space allocation and cost-benefit replication.

7 Acknowledgments

We would like to thank Nate Burnett, Nicholas Coleman, Tim Denehy, Barry Henrichs, Florentina Popovici, Muthian Sivathanu and Vijayan Prabhakaran from our department for their helpful discussions and comments on this paper. We are grateful for the excellent support provided by the members of the UW CSL.

We also wish to state our appreciation to Jeff Chase for his thoughtful analysis of our work throughout the development of this project. Finally, we thank our anonymous reviewers for their many helpful suggestions and Eric Brewer for his excellent and insightful shepherding, which has substantially improved both the content and the presentation of this paper.

This work is sponsored in part by NSF CCR-0092840, NSF UF00111, NGS-0103670, CCR-0133456, CCR-0098274, ITR-0086044, and ITR-0325267, DOE DE-FC02-01ER25450, the Wisconsin Alumni Research Foundation, EMC, and IBM.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, Dec 1993.

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, , and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In *Nucleic Acids Research*, pages 3389–3402, 1997.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain, CO, Dec 1995.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), NY, Oct 2003.

[6] Avery, P. et al. CMS Virtual Data Requirements. kholtman.home.cern.ch/kholtman/tmp/cmsreqsv6.ps, 2001.

[7] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, CA, Oct 1991.

[8] J. Bent, V. Venkataramani, N. Leroy, A. Roy, J. Stanley, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of High-Performance Distributed Computing (HPDC-11)*, pages 3–12, Edinburgh, Scotland, Jul 2002.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain, CO, Dec 1995.

[10] Y. Breitbart, A. Deacon, H.-J. Schek, A. P. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. *SIGMOD Record*, 22(3):23–30, 1993.

[11] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), Sep 2001.

[12] S. Chandra, M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct 1997.

[13] F. W. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, Louisiana, Feb 1999.

[14] J. S. Chase, L. E. Grit, D. E. Irwin, J. D. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 12)*, Seattle, WA, June 2003.

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[16] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, CA, Nov 1994.

[17] EDA Industry Working Group. The EDA Resource. http://www.eda.org/, 2003.

[18] D. A. Edwards and M. S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 58–70, Litchfield Park, Arizona, Dec 1989.

[19] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain, CO, Dec 1995.

[20] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 201–212, Copper Mountain, CO, Dec 1995.

[21] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, CA, Apr 2003.

[22] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, WA, Oct 1996.

[23] I. Foster and P. Avery. Petascale Virtual Data Grids for Data Intensive Science. GriPhyn White Paper, 2001.

[24] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[25] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[26] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi- Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, CA, Aug 2001.

[27] E. Gelenbe. On the Optimal Checkpoint Interval. *Journal of the ACM*, 26(2):259–270, Apr 1979.

[28] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), NY, Oct 2003.

[30] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[31] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, Oct 2000.

[32] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[33] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 80–93, Asheville, North Carolina, Dec 1993.

[34] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), Feb 1992.

[35] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, MA, Nov 2000.

[36] T. L. Lancaster. The Renderman Web Site. http://www.renderman.org/, 2002.

[37] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 342–353, Santiago, Chile, Sep 1994.

[38] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[39] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[40] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[41] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, WA, Dec 1985.

[42] Platform Computing. Improving Business Capacity with Distributed Computing. www.platform.com/industry/financial/, 2003.

[43] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin-Madison, Oct 2000.

[44] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[45] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, CA, June 2000.

[46] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[47] M. Rusinkiewicz and A. P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, pages 592–620. 1995.

[48] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[49] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[50] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, CA, Dec 1981.

[51] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, WA, Oct 1996.

[52] S. Sodergh. Mac, Lies, and Videotape. www.apple.com/hotnews/articles/2002/04/fullfrontal/, 2002.

[53] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, , and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[54] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of High-Performance Distributed Computing (HPDC-12)*, pages 152–161, Seattle, WA, June 2003.

[55] D. Thain and M. Livny. Parrot: Transparent User-Level Middleware for Data-Intensive Computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, Sep 2003.

[56] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, New Orleans, Louisiana, June 1998.

[57] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, Dec 1999.

ISBN 1-931971-19-6